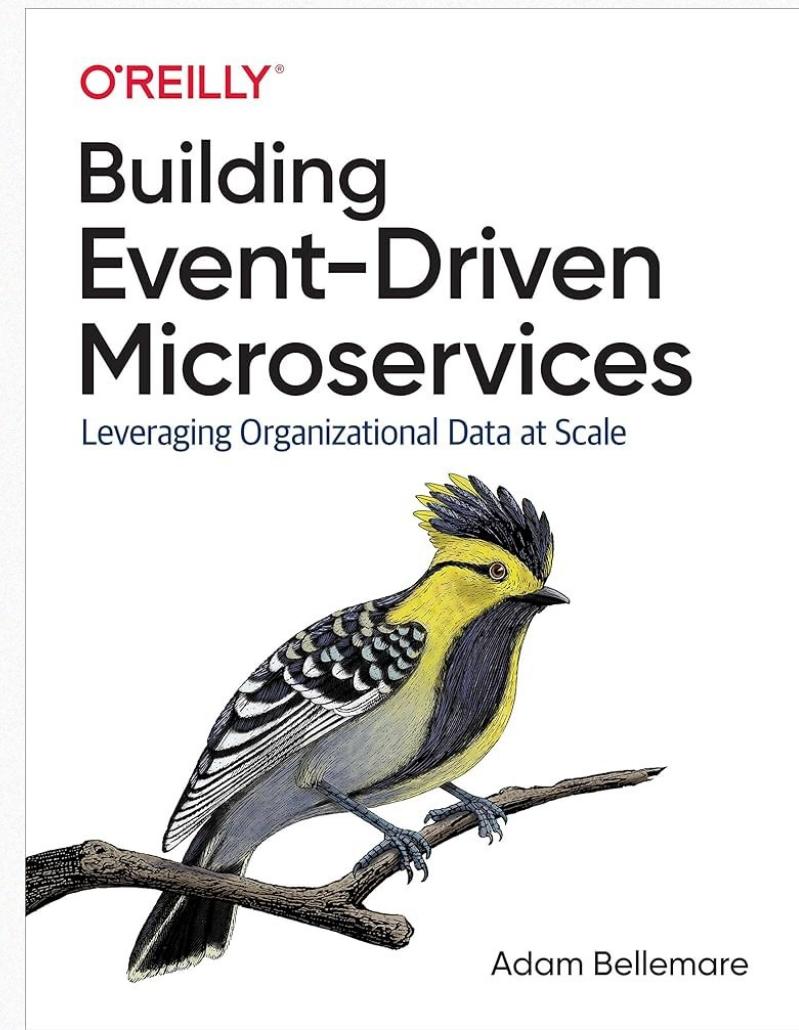
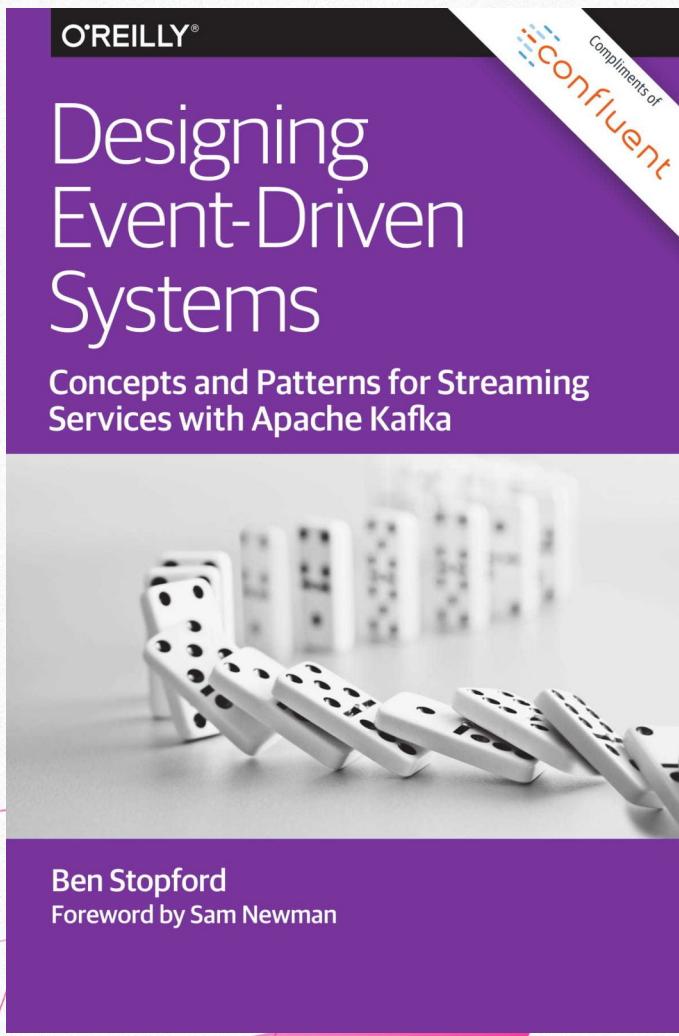


Arquitectura orientada a Eventos

“El diseño arquitectónico de software es la clave para construir sistemas sostenibles.”

— Grady Booch



¿Cómo se entera una cafetería que un cliente quiere hacer un pedido?

- **¿Necesita el mozo revisar constantemente todas las mesas?**

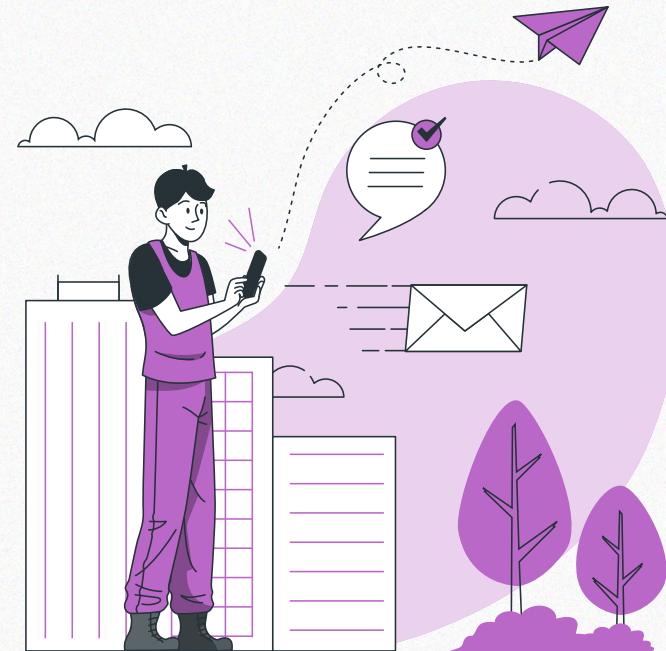
¿Y si simplemente “reaccionara” cuando alguien genera un evento?

¿Qué es un “evento”?

Introducción a Event-Driven Architecture (EDA)

La arquitectura impulsada por eventos, conocida como Event-Driven Architecture (EDA), es un paradigma de diseño que se centra en la generación, captura y procesamiento de eventos como mecanismo central para coordinar las interacciones entre componentes de un sistema.

Este enfoque se ha convertido en un pilar fundamental en la construcción de sistemas modernos, distribuidos y escalables, especialmente en contextos que requieren una alta capacidad de respuesta en tiempo real.

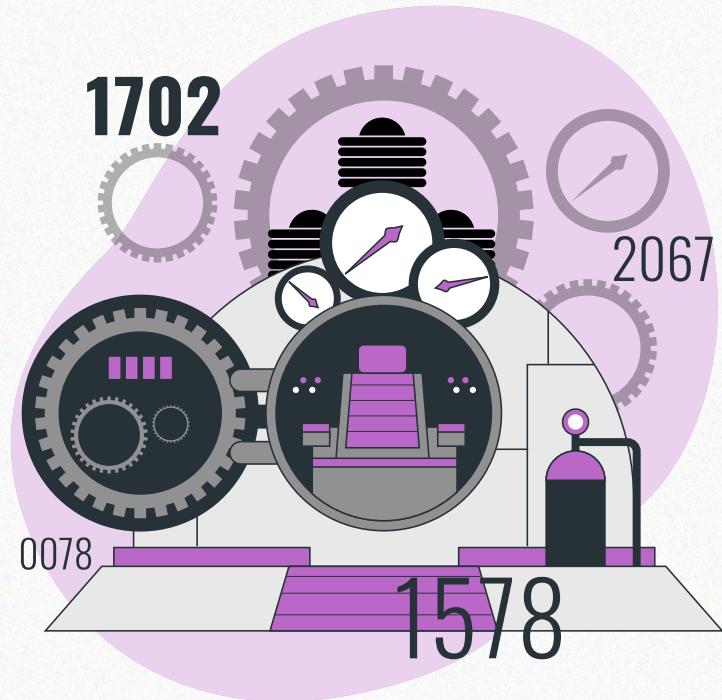


Contexto Histórico

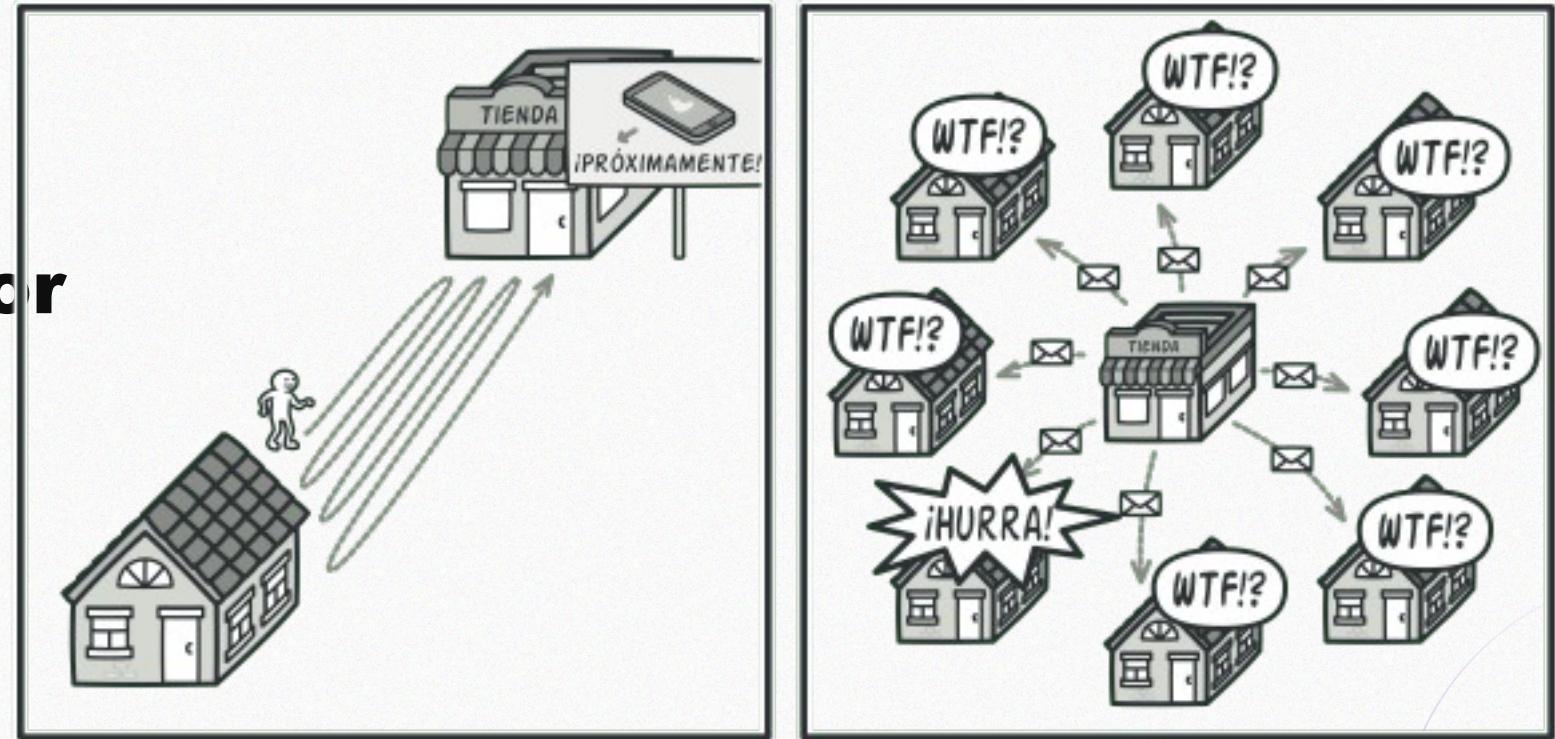
El concepto de EDA no es nuevo. En la década de 1980, los sistemas basados en eventos comenzaron a emerger en entornos de programación reactiva y en sistemas distribuidos.

La evolución de herramientas como **Java Message Service (JMS)** y, más recientemente, plataformas como **Apache Kafka**, ha impulsado su adopción en sistemas empresariales de gran escala.

Según Chen et al. (2019), EDA representa un cambio fundamental hacia sistemas más flexibles y resilientes, especialmente en comparación con arquitecturas monolíticas tradicionales.



Patrón Observador



Principios Clave de EDA

1. Eventos como ciudadanos de primera clase:

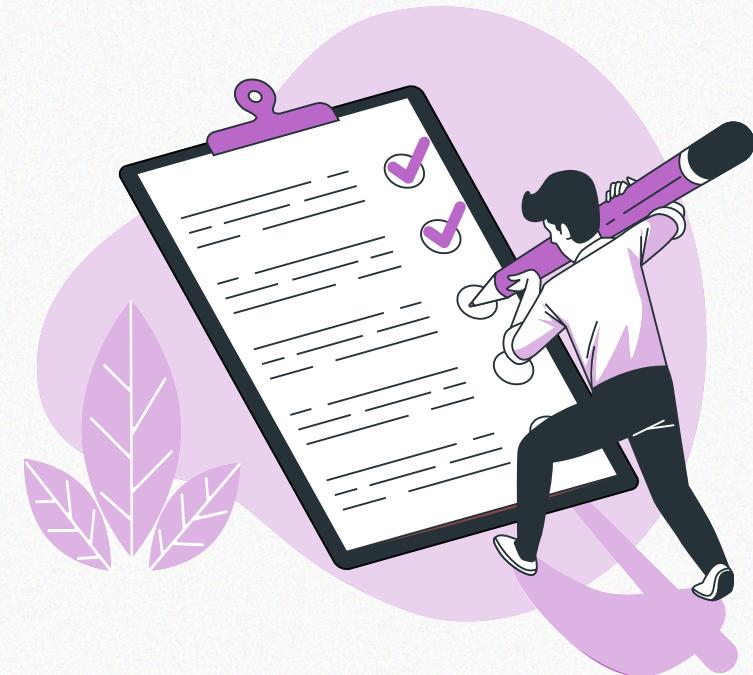
Un evento es cualquier cambio significativo en el estado de un sistema o dominio. Los eventos contienen:

- **Payload:** Datos relevantes al cambio.
- **Metadatos:** Información adicional, como la hora y la fuente del evento.

2. Desacoplamiento:

En EDA, los emisores (producers) y receptores (consumers) de eventos están desacoplados. Esto significa que no necesitan conocer la existencia o el estado del otro. El desacoplamiento permite que los componentes evolucionen de forma independiente.

- Ejemplo: En un sistema de e-commerce, un evento de “pedido creado” no necesita saber cómo se procesará el inventario.



Principios Clave de EDA

3. Procesamiento asíncrono:

Los eventos se procesan de manera asíncrona, permitiendo que el sistema reaccione en tiempo real sin bloquear otras operaciones. Esto contrasta con arquitecturas síncronas como RESTful APIs.

4. Resiliencia y escalabilidad:

Al distribuir la carga entre múltiples consumidores y garantizar la persistencia de eventos, EDA mejora la tolerancia a fallos y permite la escalabilidad horizontal (Stopford, 2017).



Comparación con Arquitecturas Tradicionales

Característica	Monolítica	SOA	EDA
Acoplamiento	Alto	Moderado	Bajo
Comunicación	Directa	Síncrona (SOAP/REST)	Asíncrona (eventos)
Escalabilidad	Limitada	Moderada	Alta
Resiliencia	Baja	Moderada	Alta

Componentes principales de Event-Driven Architecture

(EDA)

En una arquitectura impulsada por eventos (EDA), el diseño se estructura alrededor de la producción, emisión y consumo de eventos, que son los elementos fundamentales que conectan los componentes de un sistema distribuido.

Estos componentes trabajan en conjunto para garantizar el flujo continuo de información, el procesamiento eficiente y la resiliencia del sistema.



Eventos

Un evento es cualquier cambio significativo de estado en el sistema. Los eventos son mensajes transportados entre los componentes, y generalmente contienen dos partes principales:

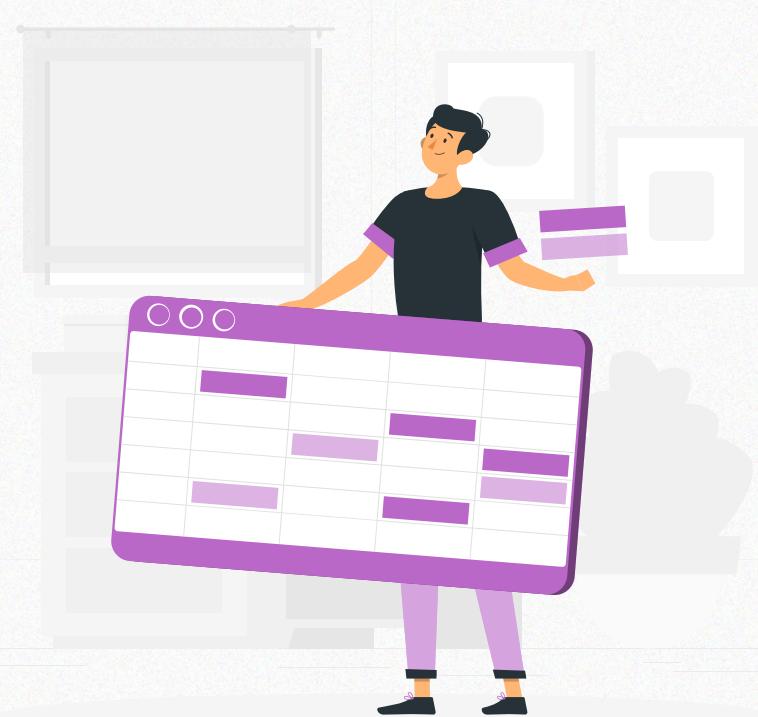
- Payload:** Representa los datos relevantes al evento, como el identificador de un pedido o un cambio en el inventario.
- Metadatos:** Proporcionan información adicional sobre el evento, como la hora de emisión, la fuente y el tipo.

```
event = {  
  "type": "OrderCreated",  
  "payload": {  
    "order_id": 123,  
    "customer_id": 456  
  },  
  "date": "2025-01-13"  
}
```

Eventos

Clasificación de eventos:

- **Eventos de dominio:** Representan cambios en el estado del dominio, como “OrderPlaced” o “UserRegistered”.
- **Eventos de integración:** Usados para comunicar información entre servicios.
- **Eventos de comandos:** Solicitudes para realizar una acción específica, como “SendNotification”.



Productores

Generan eventos y los envían al sistema. Un productor podría ser un servicio de pedidos que emite un evento OrderCreated cada vez que se realiza un nuevo pedido.

Consumidores

Reciben eventos y realizan acciones basadas en ellos. Por ejemplo, un servicio de inventario que reduce las unidades disponibles en respuesta al evento OrderCreated.

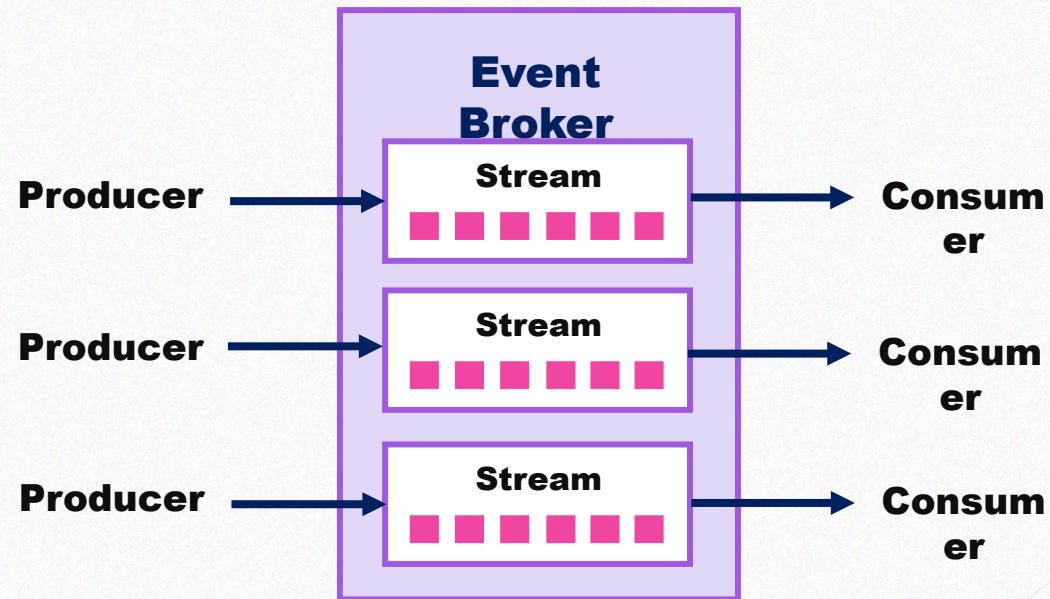
El productor **no sabe** quién consumirá el evento.
Esto permite **desacoplamiento** y mayor **flexibilidad**.



Event Brokers

El **event broker** actúa como intermediario entre productores y consumidores, asegurando que los eventos sean entregados de manera eficiente y confiable. Entre sus responsabilidades principales están:

- **Persistencia de eventos:** Almacenar eventos hasta que los consumidores estén listos para procesarlos.
- **Enrutamiento:** Dirigir eventos específicos a los consumidores interesados.
- **Escalabilidad:** Manejar grandes volúmenes de eventos distribuidos.



Los tres estilos de Event-Driven Architecture

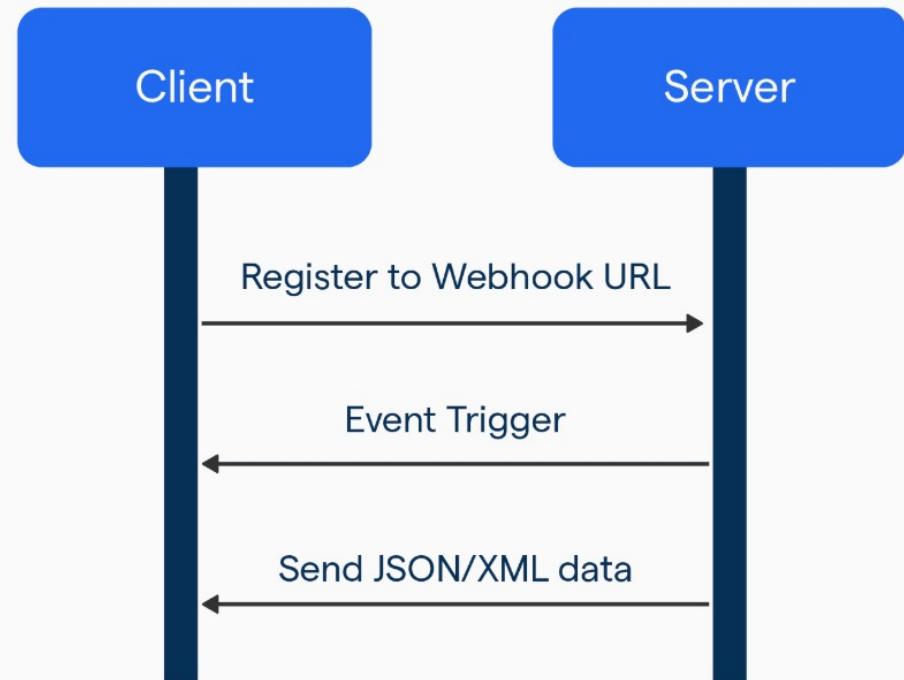
Event Notification (Notificación de evento)

Algo pasó. No te diré más. Averígualo tú.

Características

- Evento mínimo → solo el tipo (UserCreated, OrderPaid)
- El consumidor debe **preguntar** al productor por más info

```
{ "event_type": "OrderCreated", "order_id": "12345" }
```



Supongamos que recibimos un webhook que dice:

“Tu pago fue procesado con éxito”.

¿Deberíamos **creer** lo que viene en el webhook?

¿Cómo sabemos si ese webhook fue realmente enviado por el
procesador de pagos?

¿Y si alguien envía manualmente una petición POST similar
desde Postman o curl?

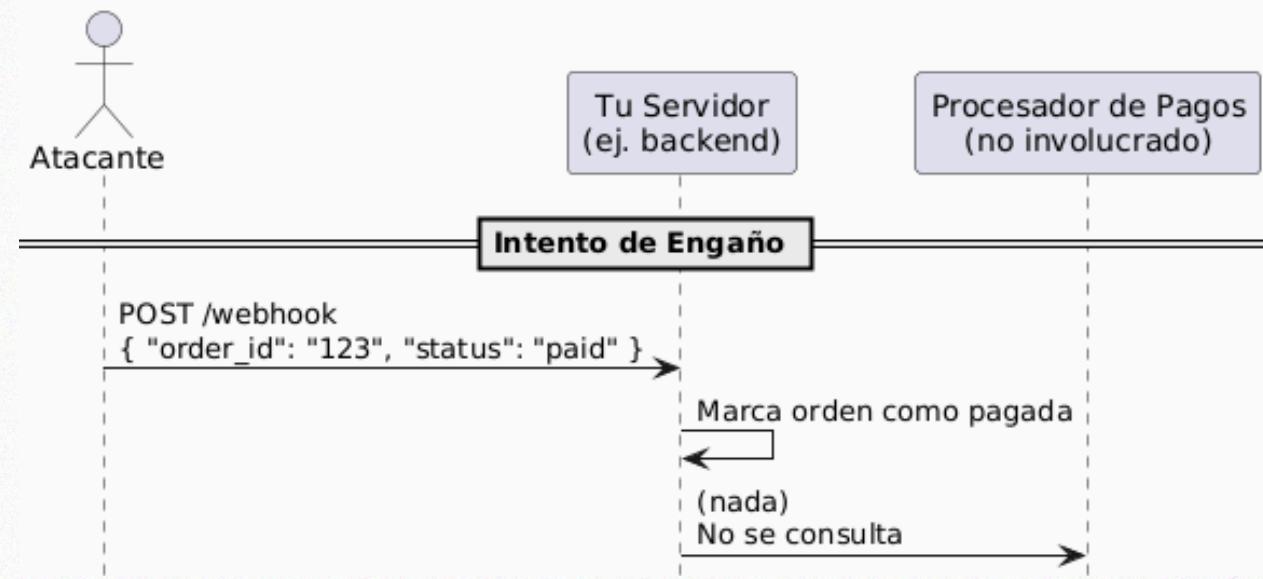
Modelo Inseguro:

Confiar ciegamente en el payload del webhook

Anti-patrón: Event-Carried State Transfer sin verificación

“El webhook me dice que el pago fue exitoso. Genial, activo la orden.”

- ¿Qué daño podría causar si este pedido activa un despacho físico?
- ¿Basta con validar la firma del webhook?
- ¿Y si interceptaron una firma válida vieja?



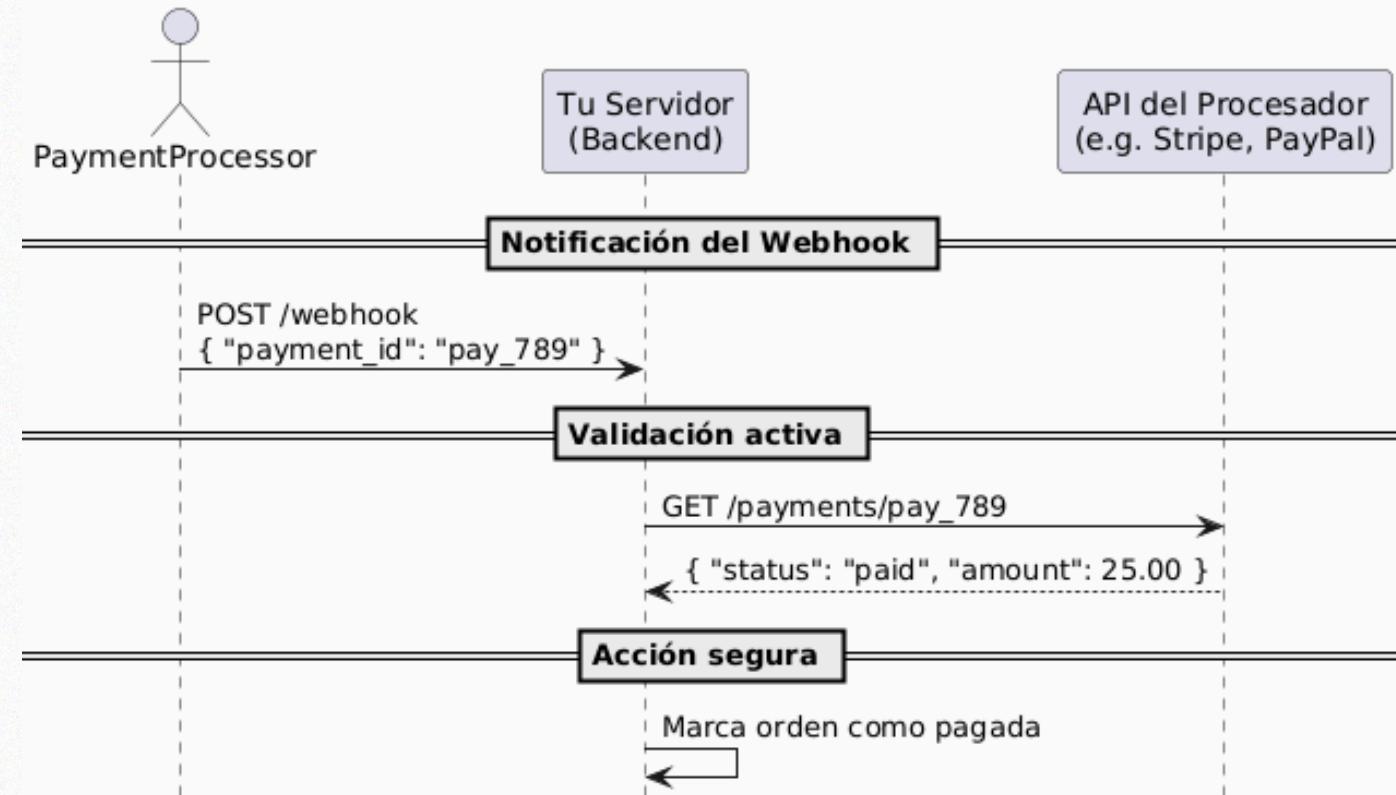
Modelo Seguro:

Event Notification + Consulta directa

“El webhook me avisa que algo pasó. Pero yo lo confirmo preguntando al procesador.”

Patrón correcto: Event Notification + Verificación del estado

El webhook sólo contiene un payment_id. Tu backend consulta al procesador de pagos y verifica el estado actual.



Modelo Seguro:

Event Notification + Consulta directa

“El webhook me avisa que algo pasó. Pero yo lo confirmo preguntando al procesador.”

Patrón correcto:

Event Notification + Verificación del estado

El webhook sólo contiene un payment_id. Tu backend consulta al procesador de pagos y verifica el estado actual.

```
from flask import Flask, request, abort
import requests

app = Flask(__name__)

PAYMENT_API_URL = "https://api.stripe.com/v1/payment_intents/"
API_KEY = "sk_test_..."

@app.route('/webhook', methods=['POST'])
def payment_webhook():
    data = request.json
    payment_id = data.get("payment_id")

    if not payment_id:
        abort(400, "Missing payment ID")

    # Verificamos el estado real consultando la API del procesador
    response = requests.get(
        f"{PAYMENT_API_URL}{payment_id}",
        headers={"Authorization": f"Bearer {API_KEY}"}
    )

    if response.status_code != 200:
        abort(502, "Payment API failed")

    payment_data = response.json()
    if payment_data.get("status") == "succeeded":
        # Confirmamos la orden
        mark_order_as_paid(payment_data["metadata"]["order_id"])
        return "OK", 200
    else:
        return "Ignored", 202

def mark_order_as_paid(order_id):
    # Actualiza en DB local
    print(f"✅ Orden {order_id} confirmada como pagada")

if __name__ == '__main__':
    app.run()
```

Event-Carried State Transfer

(Evento con estado)

Si el evento ya incluye toda la información que necesito... ¿por qué debería consultar a otro servicio?

Desacoplar componentes: El consumidor puede reaccionar al evento **sin tener que llamar al productor**.

- **Alta tolerancia a fallos:** si el productor muere, el consumidor aún puede trabajar.
- **Rendimiento:** se evitan llamadas HTTP o RPC extra.
- **Desacoplamiento:** el consumidor no necesita conocer al productor.

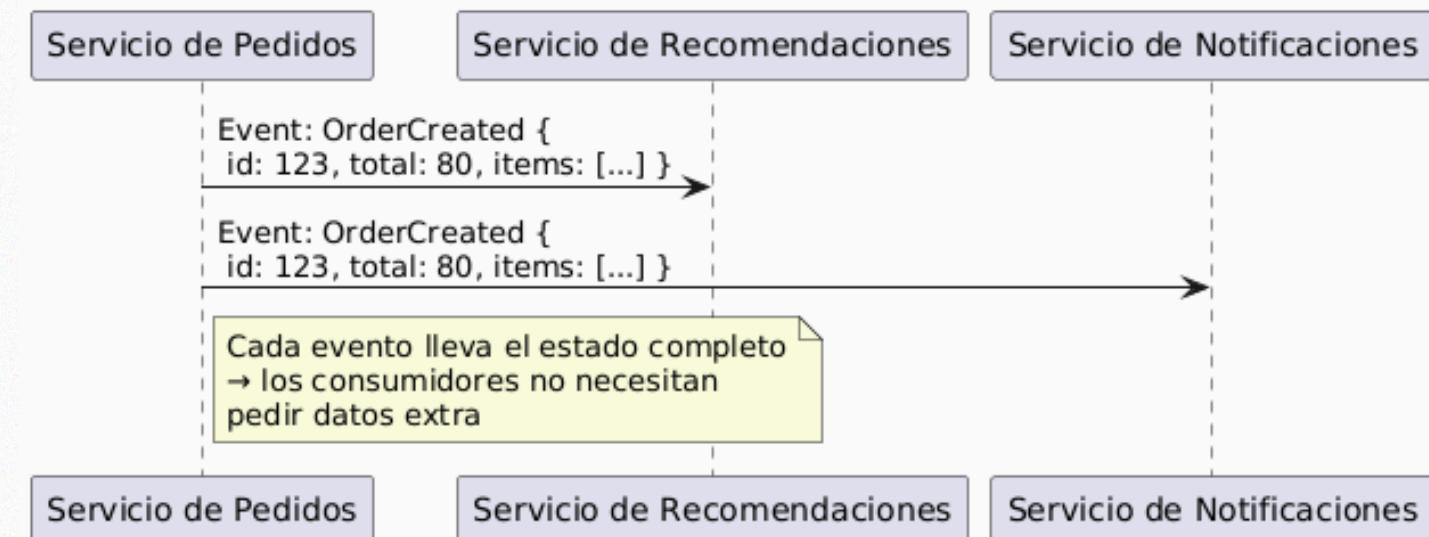
```
{  
  "event_type": "OrderCreated",  
  "order": {  
    "id": "123",  
    "items": [ "A", "B" ],  
    "total": 80,  
    "client": {  
      "id": "789",  
      "email": "juan@example.com"  
    }  
  }  
}
```

Event-Carried State Transfer (Evento con estado)

Si el evento ya incluye toda la información que necesito... ¿por qué debería consultar a otro servicio?

Desacoplar componentes: El consumidor puede reaccionar al evento **sin tener que llamar al productor.**

- **Alta tolerancia a fallos:** si el productor muere, el consumidor aún puede trabajar.
- **Rendimiento:** se evitan llamadas HTTP o RPC extra.
- **Desacoplamiento:** el consumidor no necesita conocer al productor.

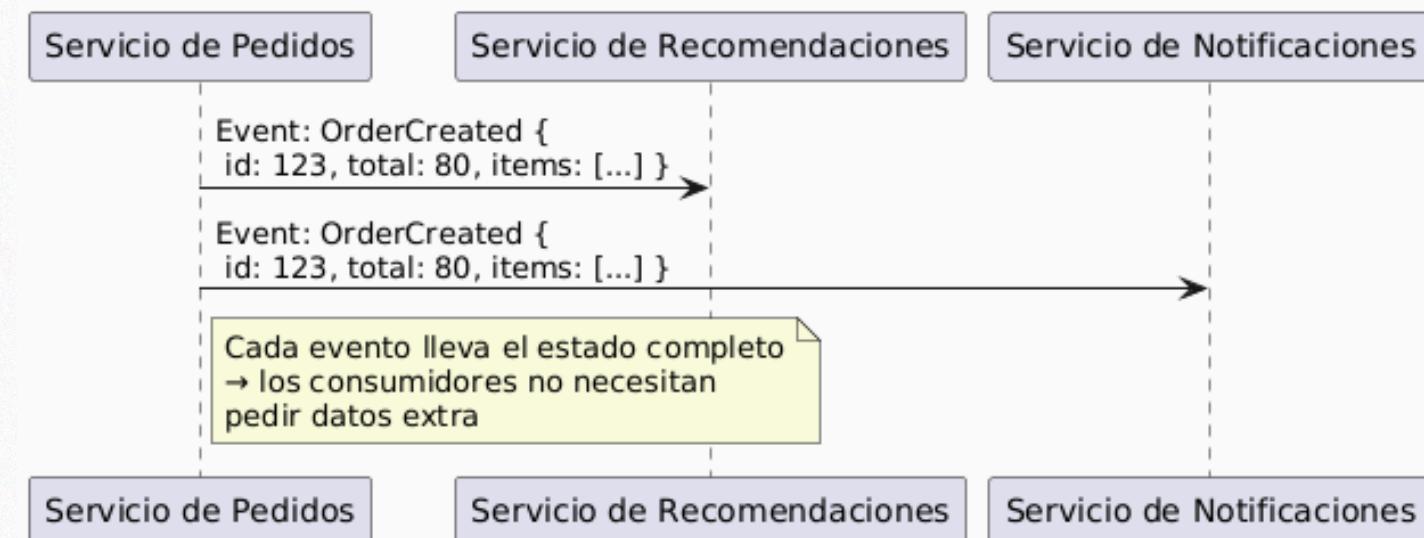


Event-Carried State Transfer (Evento con estado)

Si el evento ya incluye toda la información que necesito... ¿por qué debería consultar a otro servicio?

Buenas prácticas

- Usa **event schemas versionados** (e.g., con Avro o JSON Schema).
- Mantén los eventos **inmutables**.
- Documenta claramente qué campos se garantizan y cuáles son opcionales.
- Considera que los consumidores **persisten y cachean** esos datos.



Event Sourcing

El patrón **Event Sourcing** plantea que todos los cambios en el estado de un sistema deben registrarse como una serie de eventos inmutables.

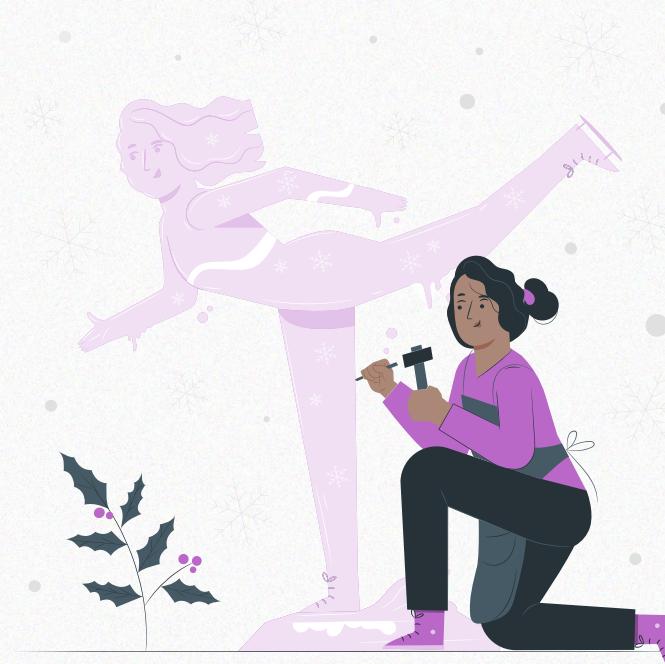
Estos eventos se convierten en la fuente única de verdad, permitiendo reconstruir el estado del sistema en cualquier momento a partir de esta secuencia.



Funcionamiento:

1. Los eventos representan cada cambio en el estado del dominio (por ejemplo, “Pedido creado”, “Producto agregado al carrito”).
2. Los eventos se almacenan en un **Event Store** como registros inmutables.
3. El estado actual del sistema se reconstruye aplicando todos los eventos en secuencia.

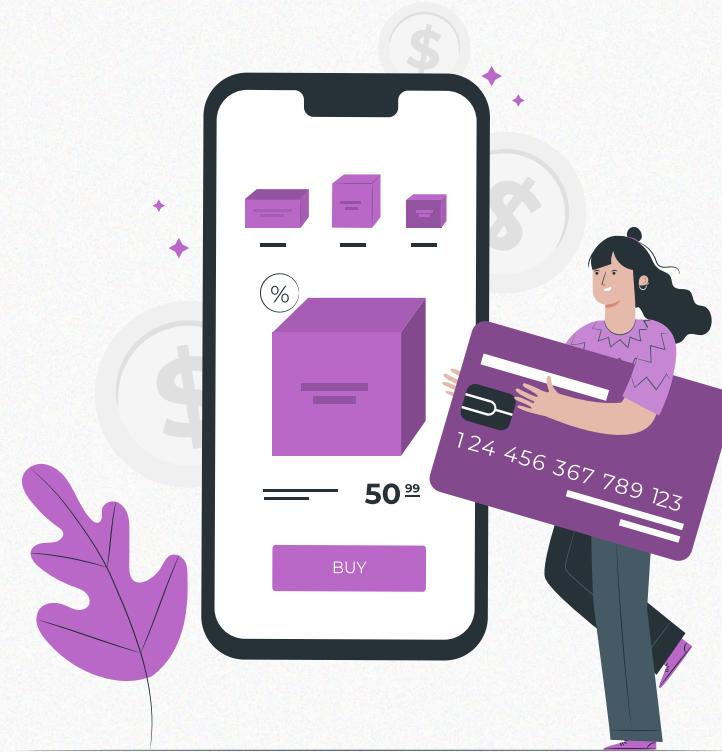
ID	Aggregate ID	Event Type	Timestamp	Data
1	order-123	OrderCreated	2025-01-14T10:00:00	{"customer_id": "cust-001", "total": 0}
2	order-123	ProductAddedToOrder	2025-01-14T10:05:00	{"product_id": "prod-101", "quantity": 2, "price": 50}
3	order-123	ProductAddedToOrder	2025-01-14T10:07:00	{"product_id": "prod-102", "quantity": 1, "price": 100}
4	order-123	OrderCompleted	2025-01-14T10:10:00	{"payment_method": "credit_card"}



Ejemplo:

- **Evento:** DepositMade con un payload de {amount : 100, accountId: 123}.
- Todos los depósitos y retiros se registran como eventos.
- El saldo actual de la cuenta se calcula aplicando estos eventos acumulativos.

EventID	EventType	Timestamp	Payload
1	DepositMade	2025-01-01 10:00:00	{"amount": 100, "accountId": 123}
2	WithdrawalMade	2025-01-02 15:30:00	{"amount": 50, "accountId": 123}
3	DepositMade	2025-01-03 09:15:00	{"amount": 200, "accountId": 123}



Beneficios:

- Reversión de cambios fácilmente (rollbacks).
- Alta trazabilidad para auditorías.
- Simplificación en la integración con otros sistemas al publicar eventos.

Desafíos:

- Complejidad en la implementación inicial.
- Necesidad de herramientas para gestionar y consultar grandes volúmenes de eventos.

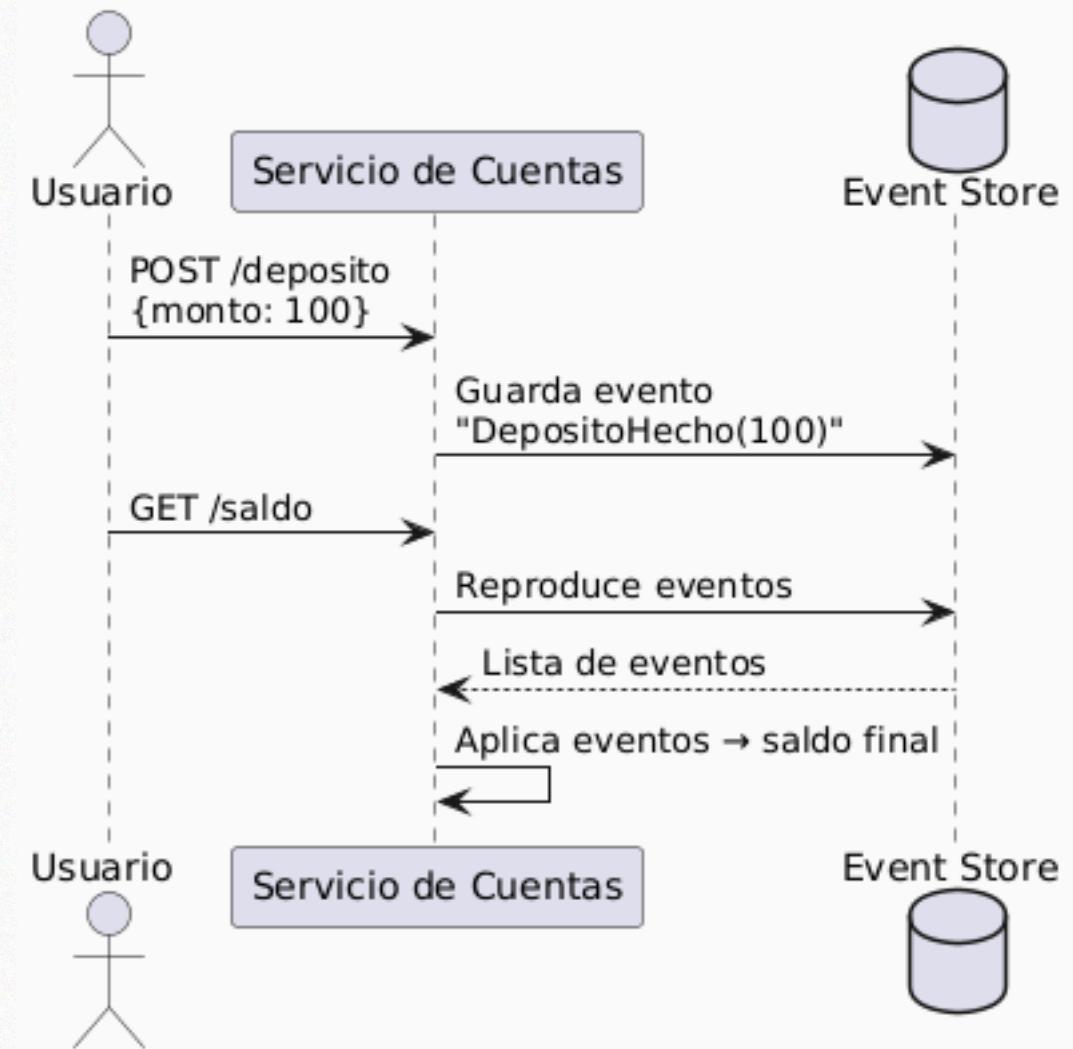


Event Sourcing

En vez de guardar el **estado final** de un objeto, se guarda **cada cambio** como un evento.

El estado actual se reconstruye *reproducido* esa secuencia de eventos.

- **Auditabilidad completa:** puedes ver qué pasó, cuándo y por qué.
- **Debug histórico:** puedes reproducir bugs.
- **Rollback o reconstrucción:** puedes rearmar el estado a cualquier punto en el tiempo.



Streams

Aspecto	Collection	Stream
📦 Almacenamiento	Sí (datos en memoria)	No (flujo de datos, no guarda datos)
🔄 Reutilizable	Sí, puedes iterar muchas veces	No, solo una vez (consumo único)
🧠 Evaluación	Eager (procesa todo de una vez)	Lazy (procesa bajo demanda)
⚙️ Operaciones	Imperativas	Declarativas (map, filter, reduce)
🤝 Paralelismo	Manual y complejo	Fácil con .parallelStream()

```
List<String> nombres = List.of("Ana", "Luis", "Pedro");

// Collection: Acceso directo
System.out.println(nombres.get(1)); // "Luis"

// Stream: Procesamiento
nombres.stream()
    .filter(n -> n.startsWith("A"))
    .forEach(System.out::println); // "Ana"
```

Streams en Java

Un **stream** en Java representa una **secuencia de datos**, que se puede recorrer, transformar, filtrar y recolectar, de forma **funcional, perezosa y potencialmente infinita**.

¿Podrías hacer lo mismo si en vez de números recibieras una secuencia infinita de eventos financieros?

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5);  
  
int result = numbers.stream()  
    .filter(n -> n % 2 == 0)  
    .map(n -> n * 10)  
    .reduce(0, Integer::sum);  
  
System.out.println(result); // Output: 60
```

```
import java.util.stream.Stream;

public class StreamProducersConsumers {
    public static void main(String[] args) {
        // Producer: Genera números consecutivos
        Stream<Integer> numbers = Stream.iterate(1, n -> n + 1);

        // Consumer para números pares
        Stream<Integer> evenNumbers = numbers.filter(n -> n % 2 == 0)
            .limit(10);

        // Consumer para números impares
        Stream<Integer> oddNumbers = Stream.iterate(1, n -> n + 1)
            .filter(n -> n % 2 != 0)
            .limit(10);

        // Procesar números pares
        System.out.println("Pares:");
        evenNumbers.forEach(System.out::println);

        // Procesar números impares
        System.out.println("\nImpares:");
        oddNumbers.forEach(System.out::println);
    }
}
```

Imagina que tienes una cinta transportadora en una fábrica de frutas   .

¿Los datos (las frutas) están almacenados en la cinta?

¿O simplemente pasan por ella mientras tú eliges cuáles procesar?

¿Puedes devolver una fruta que ya pasó?

¿Y si vienen millones de frutas, necesitas una caja para guardarlas todas antes de trabajar?

Streams en Java



Un **stream** en Java es una secuencia de elementos que se procesan de forma funcional y que permite realizar operaciones como transformación, filtrado y agregación. Los streams son útiles en EDA para manejar flujos de datos continuos.

```
Stream<String> frutas = Stream.of("🍎", "🍋", "🍊");  
  
frutas  
    .filter(f -> !f.equals("🍋"))  
    .map(f -> f + " - lavada")  
    .forEach(System.out::println);  
  
// Resultado:  
// 🍎 - lavada  
// 🍊 - lavada
```

Sintaxis Funcional y Lambdas

Sintaxis Funcional y Lambdas

Si Java es un lenguaje orientado a objetos... ¿por qué parece que estamos usando funciones como en JavaScript o Python?

Desde Java 8, puedes tratar funciones como objetos gracias a las **interfaces funcionales** y las **lambdas**.

$$f(x) = 3 + x$$



Cuerpo de la función

Parametros

¿Qué es una expresión lambda?

Es una forma breve de escribir una **función anónima**
(es decir, *sin nombre*, como una flecha: $x \rightarrow x + 1$)

$$f(x) = 3 + x$$

$$x \rightarrow 3 + x$$

`(String s) -> s.length()`

“una función que recibe un String y devuelve su longitud”.

*Es una forma compacta de representar
una función.*

Pero... aún no podemos usarla así sola.

¿Qué es una expresión lambda?

Una **interfaz funcional** es una interfaz que contiene exactamente un único método abstracto.

¿Por qué nace la interfaz funcional?

Java sigue siendo fuertemente tipado, así que necesita **una forma de tipar las funciones** si las queremos usar como ciudadanos de primer orden.

Esto se conoce como “**Single Abstract Method interfaces**” (SAM interfaces).

La lambda es una forma de implementar esa interfaz **implícitamente**, sin usar new.

```
@FunctionalInterface  
interface Funcion {  
    int aplicar(int x);  
}
```

Funcion f = x -> x + 1;

Ahora, la lambda tiene un **tipo concreto** y válido en el sistema de tipos de Java.

¿Qué es f ahora?
¿Qué tipo tiene?
¿Qué hace?

¿Qué es una expresión lambda?

Una **interfaz funcional** es una interfaz que contiene exactamente un único método abstracto.

¿Por qué nace la interfaz funcional?

Java sigue siendo fuertemente tipado, así que necesita **una forma de tipar las funciones** si las queremos usar como ciudadanos de primer orden.

Esto se conoce como “**Single Abstract Method interfaces**” (SAM interfaces).

La lambda es una forma de implementar esa interfaz **implícitamente**, sin usar new.

```
Funcion f = new Funcion() {  
    public int aplicar(int x) {  
        return x + 1;  
    }  
};
```

```
public class Main {  
    @FunctionalInterface  
    interface Funcion {  
        int aplicar(int x);  
    }  
  
    public static void main(String[] args) {  
        Funcion sumarUno = x -> x + 1;  
  
        int resultado = sumarUno.aplicar(10);  
        System.out.println("Resultado: " + resultado);  
    }  
}
```

Guardar y pasar lambdas como parámetros

Primero, definamos una interfaz funcional:

```
@FunctionalInterface
interface Funcion {
    int aplicar(int x);
}
```

Creamos varias lambdas compatibles con esa interfaz:

```
Funcion sumarTres = x -> x + 3;
Funcion duplicar = x -> x * 2;
Funcion cuadrado = x -> x * x;
```

Ahora creamos un método que reciba una lambda como argumento:

```
public static int ejecutarFuncion(Funcion f, int valor) {
    return f.aplicar(valor);
}
```

```
public class Main {
    public static void main(String[] args) {
        Funcion sumarTres = x -> x + 3;
        Funcion duplicar = x -> x * 2;

        int a = ejecutarFuncion(sumarTres, 5); // 8
        int b = ejecutarFuncion(duplicar, 4); // 8

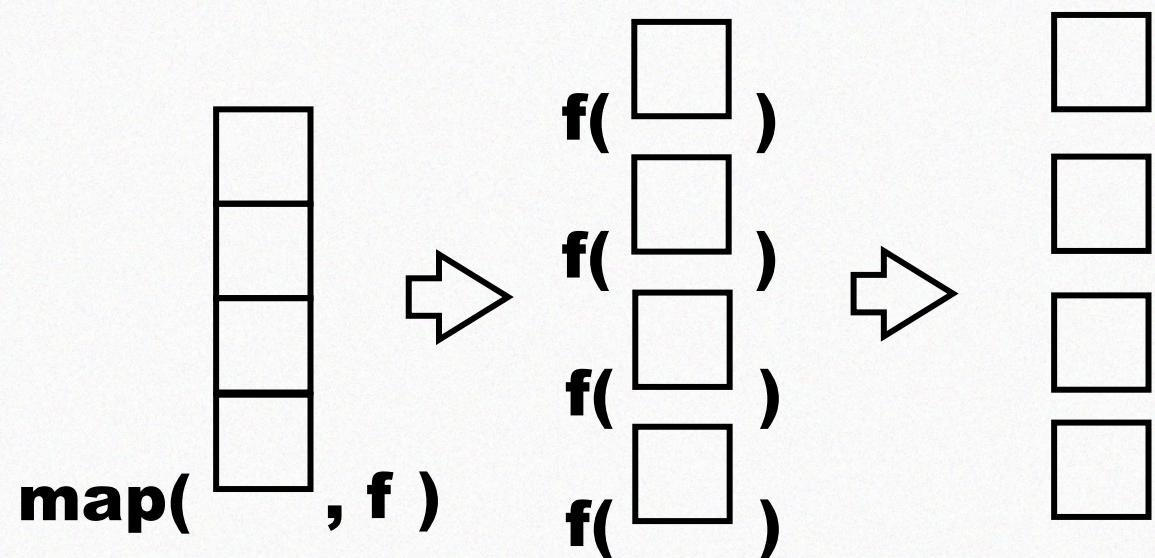
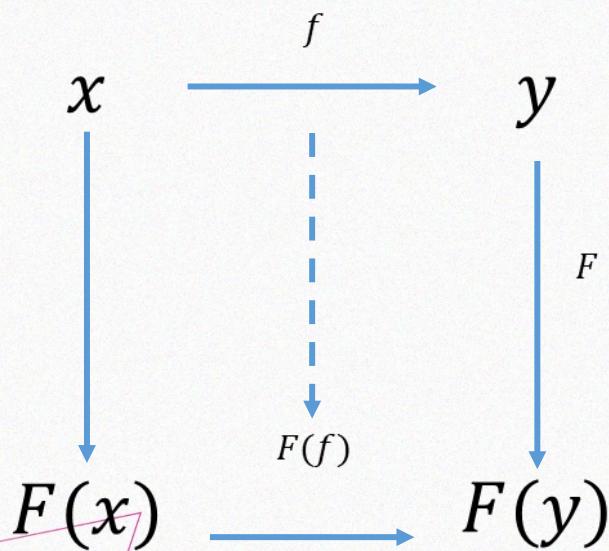
        System.out.println("Resultado A: " + a);
        System.out.println("Resultado B: " + b);
    }

    public static int ejecutarFuncion(Funcion f, int valor) {
        return f.aplicar(valor);
    }
}

@FunctionalInterface
interface Funcion {
    int aplicar(int x);
}
```

Funciones de Orden Superior

- Recibe otra función como parámetro
- Devuelve una función como resultado



Funciones de Orden Superior

- Recibe otra función como parámetro
- Devuelve una función como resultado

Los métodos de Stream como map, filter y forEach **esperan funciones**. Las lambdas **son funciones que implementan interfaces funcionales**, y por eso podemos pasárselas como argumentos sin crear clases nuevas.

Método	¿Qué hace?	¿Devuelve Stream?	Ejemplo
filter	Filtrar según condición (true o false)	<input checked="" type="checkbox"/> Sí	.filter(n -> n > 3)
map	Transforma cada elemento	<input checked="" type="checkbox"/> Sí	.map(n -> n * 2)
forEach	Ejecuta acción sobre cada elemento	<input type="checkbox"/> No	.foreach(System.out::println)

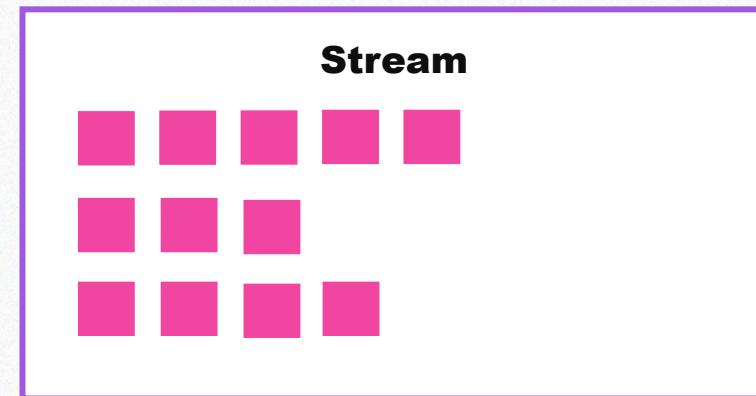
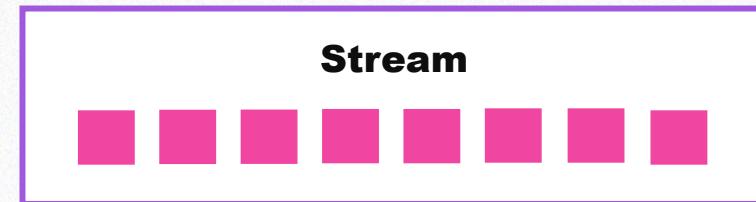
Funciones de Orden Superior

Método	¿Qué hace?	¿Devuelve Stream?	Ejemplo
reduce	Combina todos los elementos en un único valor acumulado	✗ No	.reduce(0, (a, b) -> a + b)
collect	Acumula los resultados en una colección (lista, set...)	✗ No	.collect(Collectors.toList())
sorted	Ordena los elementos	✓ Sí	.sorted()
limit	Toma solo los primeros n elementos	✓ Sí	.limit(5)
distinct	Elimina duplicados	✓ Sí	.distinct()
skip	Omite los primeros n elementos	✓ Sí	.skip(3)
count	Devuelve cuántos elementos tiene el stream	✗ No	.count()
anyMatch	Verifica si algún elemento cumple una condición	✗ No	.anyMatch(n -> n > 10)
allMatch	Verifica si todos cumplen una condición	✗ No	.allMatch(n -> n > 0)
noneMatch	Verifica si ninguno cumple una condición	✗ No	.noneMatch(n -> n < 0)
findFirst	Devuelve el primer elemento (opcional)	✗ No	.findFirst().orElse(-1)

Streams en arquitecturas orientadas a eventos

Stream = una secuencia ordenada, continua e inmutable de eventos asociados a una fuente (entidad, tipo de evento, partición, etc.)

Tipo de stream	Definición	Ejemplo
Por entidad	Stream de eventos para una entidad específica	Eventos de la cuenta ACC001
Por tipo de evento	Stream agrupado por tipo de evento	Todos los OrderCreated del sistema
Por canal de negocio	Stream lógico que agrupa eventos heterogéneos relacionados	Eventos de ventas online

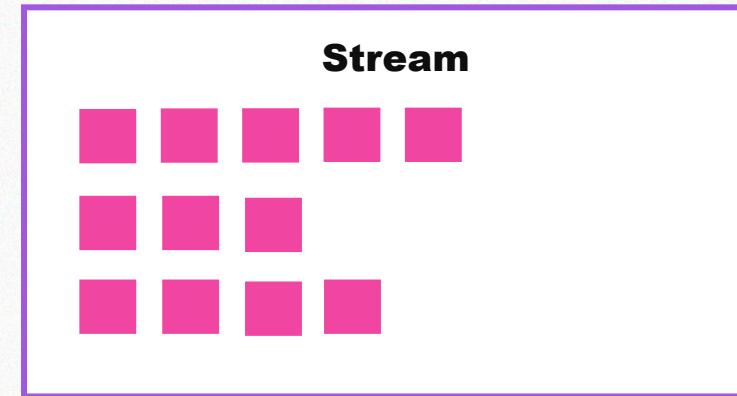


Secuenciales

Procesan los elementos de forma ordenada, uno por uno.

Paralelos

Dividen los datos en subflujos y procesan cada uno en paralelo.



Métodos de Creación de Streams

Stream.of(T... values): Crea un Stream a partir de un conjunto de valores.

```
Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5);
```

Stream.empty(): Devuelve un flujo vacío.

```
Stream<String> emptyStream = Stream.empty();
```

Stream.generate(Supplier<? extends T> s): Genera un flujo infinito basado en un Supplier.

```
Stream<Double> randomNumbers = Stream.generate(Math::random);
```

Stream.iterate(T seed, UnaryOperator<T> f): Crea un flujo infinito comenzando con un valor inicial y aplicando una función iterativa.

```
Stream<Integer> stream = Stream.iterate(0, n -> n + 2);
```

Métodos de Creación de Streams

Arrays.stream(T[] array): Crea un flujo a partir de un array.

```
int[] numbers = {1, 2, 3};  
IntStream stream = Arrays.stream(numbers);
```

Collection.stream(): Convierte una colección a un Stream.

```
List<String> list = List.of("A", "B", "C");  
Stream<String> stream = list.stream();
```

Collection.parallelStream(): Crea un flujo paralelo a partir de una colección.

```
Stream<String> parallelStream = list.parallelStream();
```



Transformaciones

map(Function<? super T, ? extends R> mapper): Aplica una función a cada elemento del flujo.

```
Stream<String> stream = Stream.of("a", "b", "c");
stream.map(String::toUpperCase).forEach(System.out::println);
```

flatMap(Function<? super T, ? extends Stream<? extends R>> mapper): Descompone elementos complejos en varios elementos simples.

```
List<List<Integer>> nestedList = List.of(List.of(1, 2),
List.of(3, 4));
nestedList.stream().flatMap(List::stream).forEach(System
.out::println);
```

filter(Predicate<? super T> predicate): Filtra los elementos que cumplen una condición.

```
Stream<Integer> stream = Stream.of(1, 2, 3, 4);
stream.filter(n -> n % 2 == 0).forEach(System.out::println);
```

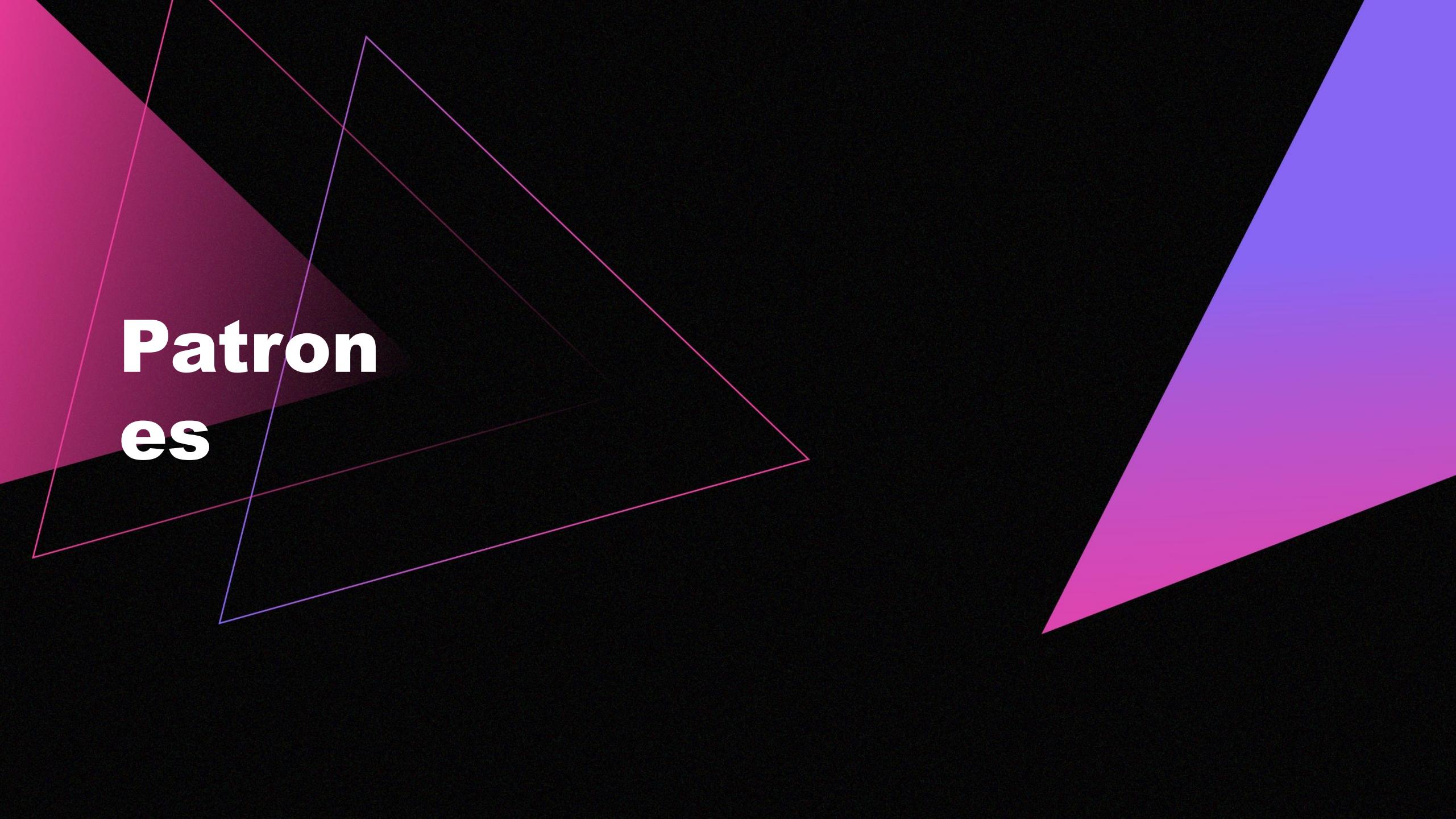


```
// Muestra el video en vivo usando el Stream
CanvasFrame canvas = new CanvasFrame("Video Stream", CanvasFrame.getDefaultGamma() / grabber

// Procesa y muestra los primeros 100 frames
frameStream.limit(100) // Limitamos a 100 frames para evitar flujo infinito
    .filter(frame -> frame != null) // Filtra frames nulos
    .map(frame -> {
        // Convierte a Mat para procesar
        Mat mat = OpenCVFrameConverter.ToMat.convert(frame);

        // Aplica operación: Escalado de grises
        Imgproc.cvtColor(mat, mat, Imgproc.COLOR_BGR2GRAY);

        // Convierte de nuevo a Frame
        return OpenCVFrameConverter.ToMat.convert(mat);
    })
    .forEach(processedFrame -> {
        if (canvas.isVisible()) {
            canvas.showImage(processedFrame);
        }
    });
}
```



The background features abstract geometric shapes. On the left, there is a large, semi-transparent red triangle pointing upwards. Overlaid on it are several thin, light blue lines forming a complex, overlapping pattern. On the right side, there is a large, semi-transparent blue triangle pointing downwards. The overall composition is minimalist and modern, using a high-contrast color palette of black, red, and blue against a white background.

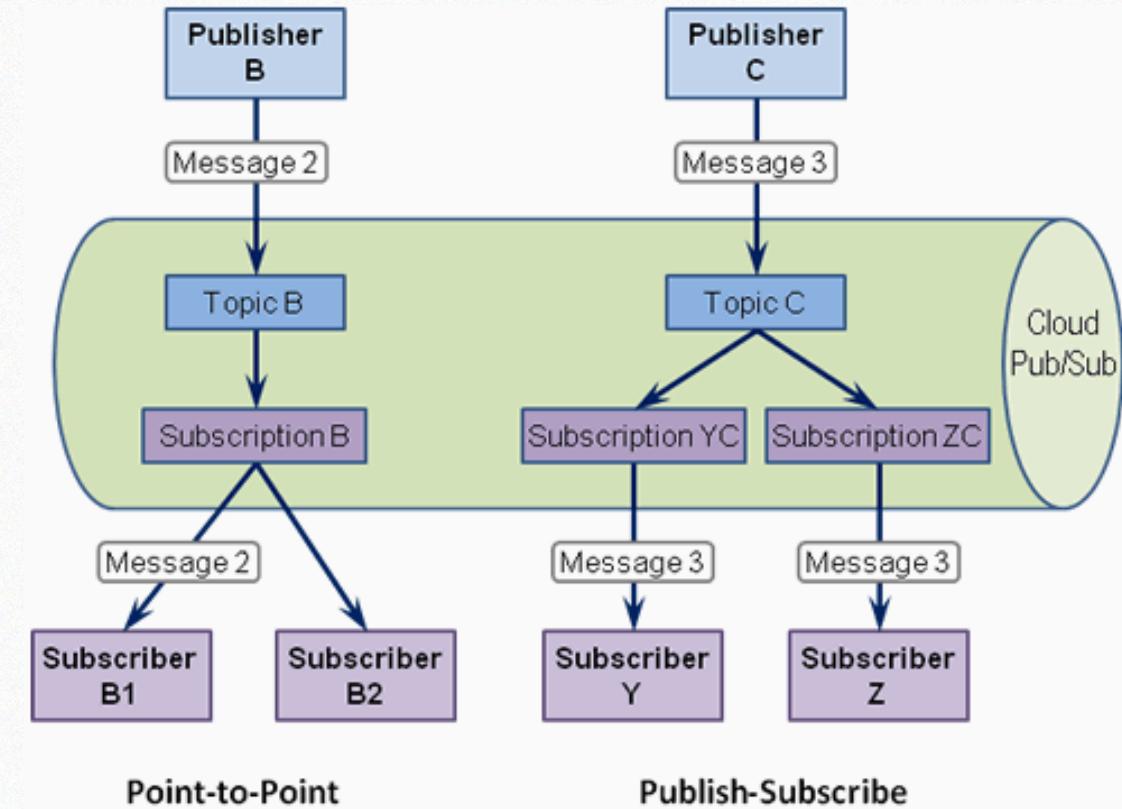
Patron es

Publish-Subscribe Pattern

El patrón Pub/Sub es un estilo de mensajería en el que los publicadores envían mensajes sin conocer a los suscriptores y viceversa.

Características principales:

- **Desacoplamiento:** Los publicadores y suscriptores no tienen conocimiento mutuo directo.
- **Intermediario:** Generalmente, un sistema de mensajería (por ejemplo, Kafka, RabbitMQ) actúa como intermediario para manejar la entrega de mensajes.
- **Escalabilidad:** Permite que múltiples suscriptores reciban eventos simultáneamente.



Beneficios:

- Alta flexibilidad y facilidad de integración.
- Eficiencia en el manejo de eventos masivos.

Desafíos:

- Manejo de la confiabilidad en la entrega de mensajes.
- Complejidad en la gestión de suscriptores y configuraciones.



¿Qué es RxJava?

RxJava es una librería para manejar flujos de datos **asíncronos y en tiempo real**, utilizando el patrón **Observador + programación funcional**.

```
import io.reactivex.rxjava3.subjects.PublishSubject;

public class Main {
    public static void main(String[] args) {
        PublishSubject<Integer> sujeto = PublishSubject.create();

        sujeto
            .map(n -> n * 2)
            .filter(n -> n > 5)
            .subscribe(n -> System.out.println("Recibo: " + n));

        // Emitimos datos dinámicamente
        sujeto.onNext(1);
        sujeto.onNext(3);
        sujeto.onNext(4); // este sí pasará el filtro (4*2=8 > 5)
        sujeto.onNext(10); // también

        sujeto.onComplete(); // opcional
    }
}
```



Paso 1: Definir el evento

```
public class ErrorInternoEvent {  
    private final String mensaje;  
    private final int codigo;  
  
    public ErrorInternoEvent(String mensaje, int codigo) {  
        this.mensaje = mensaje;  
        this.codigo = codigo;  
    }  
  
    public String getMensaje() {  
        return mensaje;  
    }  
  
    public int getCodigo() {  
        return codigo;  
    }  
}
```



Paso 2: Configurar el Sink reactivo

```
@Configuration
public class EventBusConfig {

    @Bean
    public Sinks.Many<ErrorInternoEvent> errorSink() {
        return Sinks.many().multicast().onBackpressureBuffer();
    }

    @Bean
    public Flux<ErrorInternoEvent> errorFlux(Sinks.Many<ErrorInternoEvent> errorSink) {
        return errorSink.asFlux();
    }
}
```

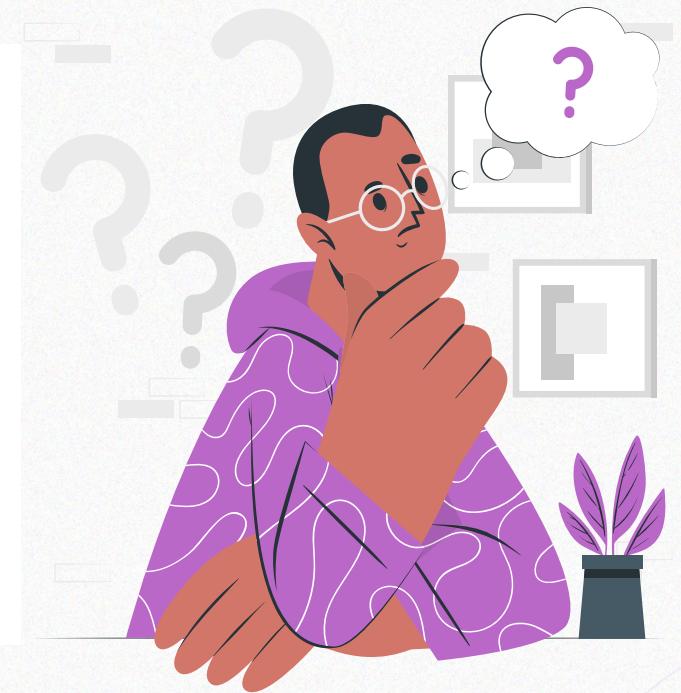
Paso 3: Publicar eventos

```
@Service
public class ErrorPublisherService {

    private final Sinks.Many<ErrorInternoEvent> sink;

    public ErrorPublisherService(Sinks.Many<ErrorInternoEvent> sink) {
        this.sink = sink;
    }

    public void reportarError(String mensaje) {
        sink.tryEmitNext(new ErrorInternoEvent(mensaje, 500));
    }
}
```



Paso 4: Escuchar el stream y actuar

```
@Service
public class ErrorSubscriberService {

    public ErrorSubscriberService(Flux<ErrorInternoEvent> errorFlux) {
        errorFlux
            .filter(event -> event.getCodigo() == 500)
            .subscribe(event -> {
                System.out.println("✉️ Enviando correo por error: " + event.getMensaje())
                // Aquí llamas a tu servicio de correo real
            });
    }
}
```



Paso 5: Disparar evento desde un controlador

```
@RestController
@RequestMapping("/test")
public class TestController {

    private final ErrorPublisherService publisher;

    public TestController(ErrorPublisherService publisher) {
        this.publisher = publisher;
    }

    @GetMapping("/error")
    public Mono<String> dispararError() {
        publisher.reportarError("Se detectó un 500 en /test/error");
        return Mono.just("Error reportado");
    }
}
```

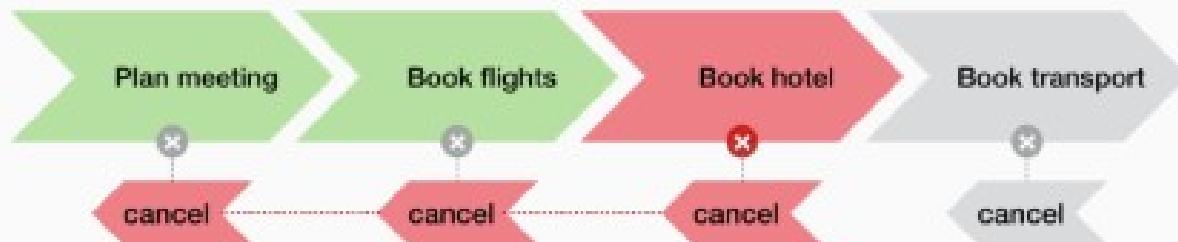


Saga Pattern:

El **Saga Pattern** es un patrón arquitectural utilizado para manejar **transacciones distribuidas** en sistemas de microservicios o distribuidos. En lugar de depender de transacciones globales que bloquean recursos en varios servicios, **Saga** divide la transacción en una serie de pasos locales y define acciones compensatorias para garantizar la consistencia eventual.

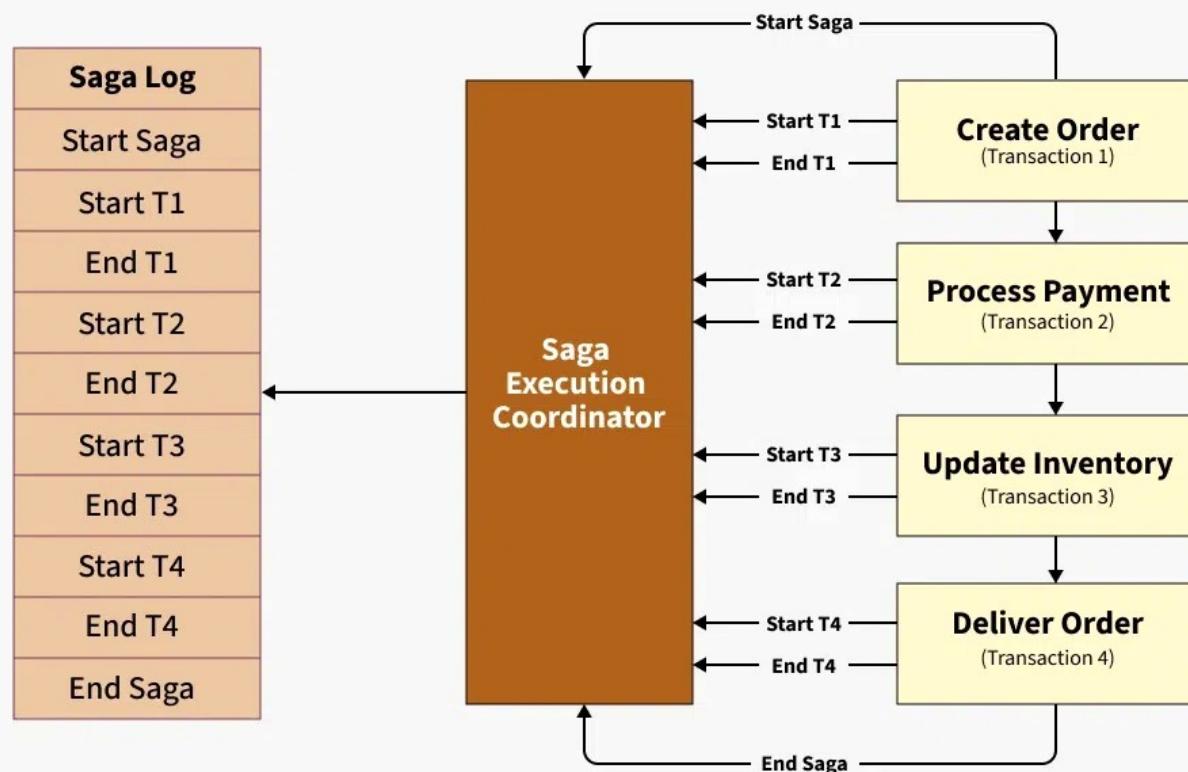
Tipos de coordinación:

- **Orquestación**: Una entidad central (orquestrador) gestiona el flujo de pasos en la saga.
- **Coreografía**: Cada servicio reacciona a eventos generados por otros servicios para avanzar en el flujo.



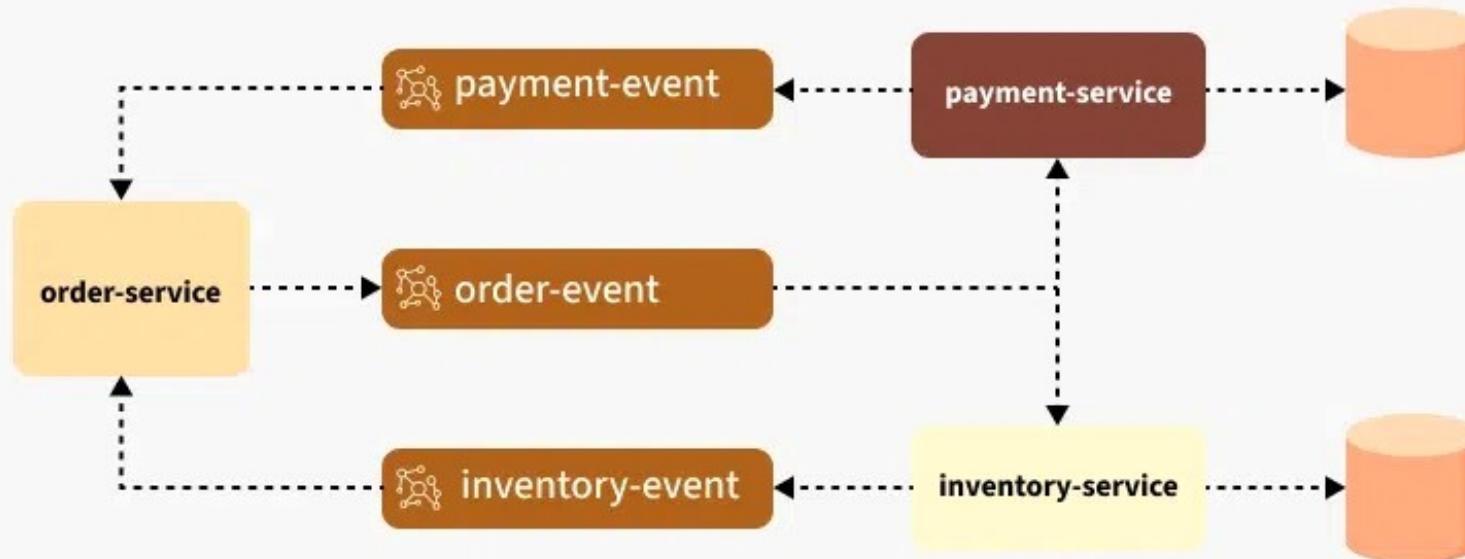
Saga Pattern:

Orchestration-Based Approach (Centralized)



Saga Pattern:

Choreography-Based Approach (Event-Driven)



Messaging Patterns

(Patrones de Mensajería)

Confirmación de mensajes (Message Acknowledgement)

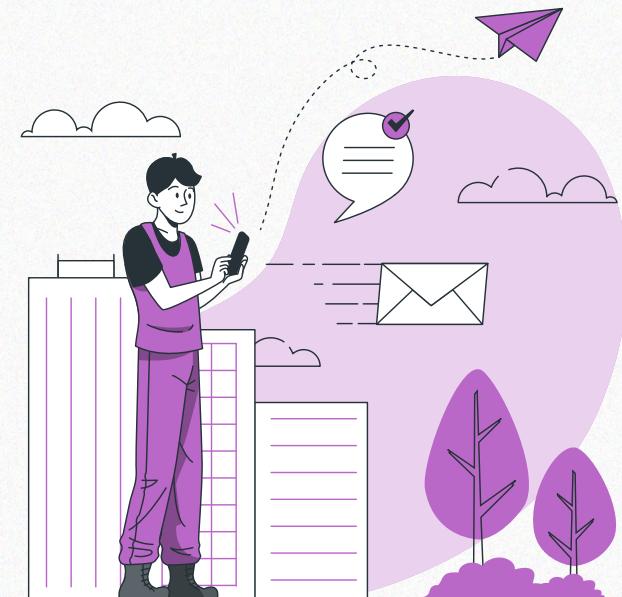
Este patrón asegura que un mensaje enviado por un productor sea procesado con éxito por el consumidor antes de considerarlo como entregado.

- **Implementación:**

- Los consumidores envían un ACK al broker después de procesar un mensaje.
- En caso de error, se puede enviar un NACK o dejar que el mensaje expire.

- **Ejemplo:**

- **RabbitMQ:** Usa basic_ack y basic_nack.
- **Kafka:** Los consumidores confirman el procesamiento mediante el commit del offset.



Implementación ACK

```
# Ejemplo de ACK en un consumidor RabbitMQ
import pika

connection = pika.BlockingConnection(pika.ConnectionParameters('local
channel = connection.channel()

# Cola de mensajes
channel.queue_declare(queue='events')

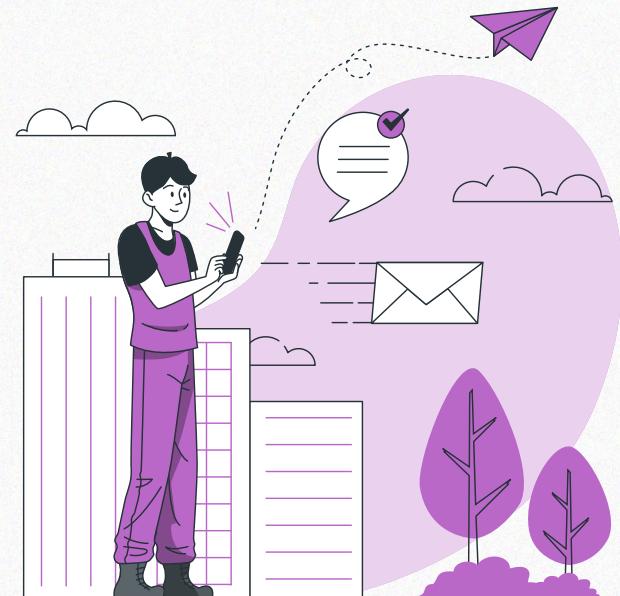
def callback(ch, method, properties, body):
    print(f'Mensaje recibido: {body}')
    try:
        # Procesamiento exitoso
        process_message(body)
        # Enviar ACK
        ch.basic_ack(delivery_tag=method.delivery_tag)
    except Exception as e:
        print(f'Error al procesar mensaje: {e}')
        # Enviar NACK (no volver a enviar)
        ch.basic_nack(delivery_tag=method.delivery_tag, requeue=False)

channel.basic_consume(queue='events', on_message_callback=callback,
auto_ack=False)
print("Esperando mensajes...")
channel.start_consuming()
```

Redelivery (Reenvío de mensajes no confirmados)

Si un consumidor no envía un ACK dentro de un tiempo definido, el mensaje se reentrega a otro consumidor o a una cola de errores.

- **Ejemplo:**
- **Dead Letter Queue (DLQ):** Los mensajes no confirmados después de varios intentos se envían a una cola de errores para su análisis posterior.



Implementación ReDelivery

```
from kafka import KafkaConsumer

consumer = KafkaConsumer(
    'my-topic',
    bootstrap_servers='localhost:9092',
    group_id='my-group',
    enable_auto_commit=False # Deshabilita el autocommit
)

for message in consumer:
    try:
        print(f"Procesando: {message.value}")
        if message.value == b'error': # Simula un error
            raise Exception("Error simulado")
        # Confirmar el offset después del procesamiento exitoso
        consumer.commit()
    except Exception as e:
        print(f"Error procesando mensaje: {e}")
        # Kafka redeliverá el mensaje porque no se confirmó el off
```

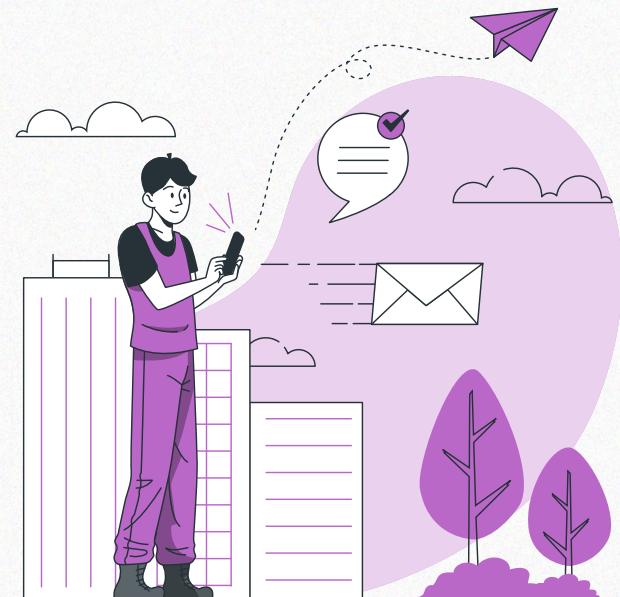
Patrón de Retry (Reintentos)

Este patrón maneja mensajes fallidos reenviándolos para su procesamiento. Los ACKs son fundamentales para decidir si un mensaje debe reenviarse.

- Si el consumidor no envía un ACK, el broker asume que falló y vuelve a enviar el mensaje.
- Los mensajes pueden reenviarse inmediatamente o después de un retraso.

- **Ejemplo:**

- RabbitMQ con colas de reintentos.
- Kafka con políticas de reintentos configuradas(`retry.backoff.ms`).



```
# Declarar el intercambio principal
channel.exchange_declare(exchange='main_exchange', exchange_type='dir')

# Declarar la cola principal
channel.queue_declare(
    queue='main_queue',
    arguments={
        'x-dead-letter-exchange': 'retry_exchange', # Enviar mensajes de error
    }
)
channel.queue_bind(exchange='main_exchange', queue='main_queue', routing_key='')

# Declarar el intercambio de reintentos
channel.exchange_declare(exchange='retry_exchange', exchange_type='direct')

# Declarar la cola de reintentos
channel.queue_declare(
    queue='retry_queue',
    arguments={
        'x-dead-letter-exchange': 'main_exchange', # Regresar mensajes de error
        'x-message-ttl': 10000 # 10 segundos de TTL
    }
)
channel.queue_bind(exchange='retry_exchange', queue='retry_queue', routing_key='')
```

Implementación Retry

Patrón de Procesamiento Exactly-Once

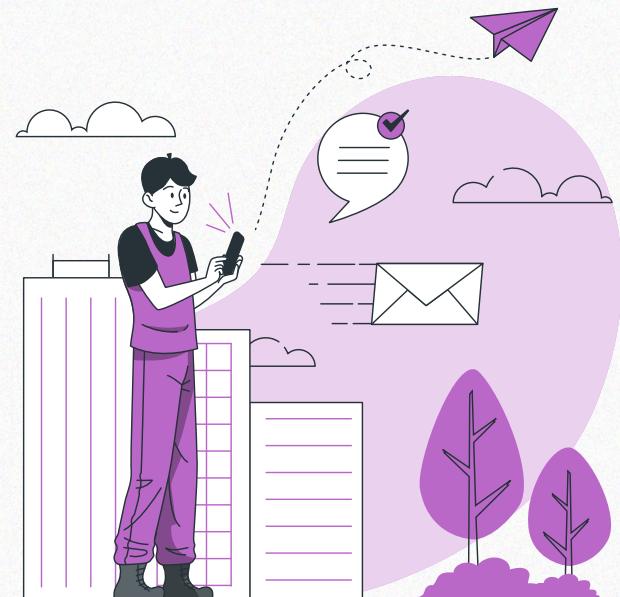
Este patrón asegura que un evento o mensaje se procese una sola vez, incluso en sistemas distribuidos.

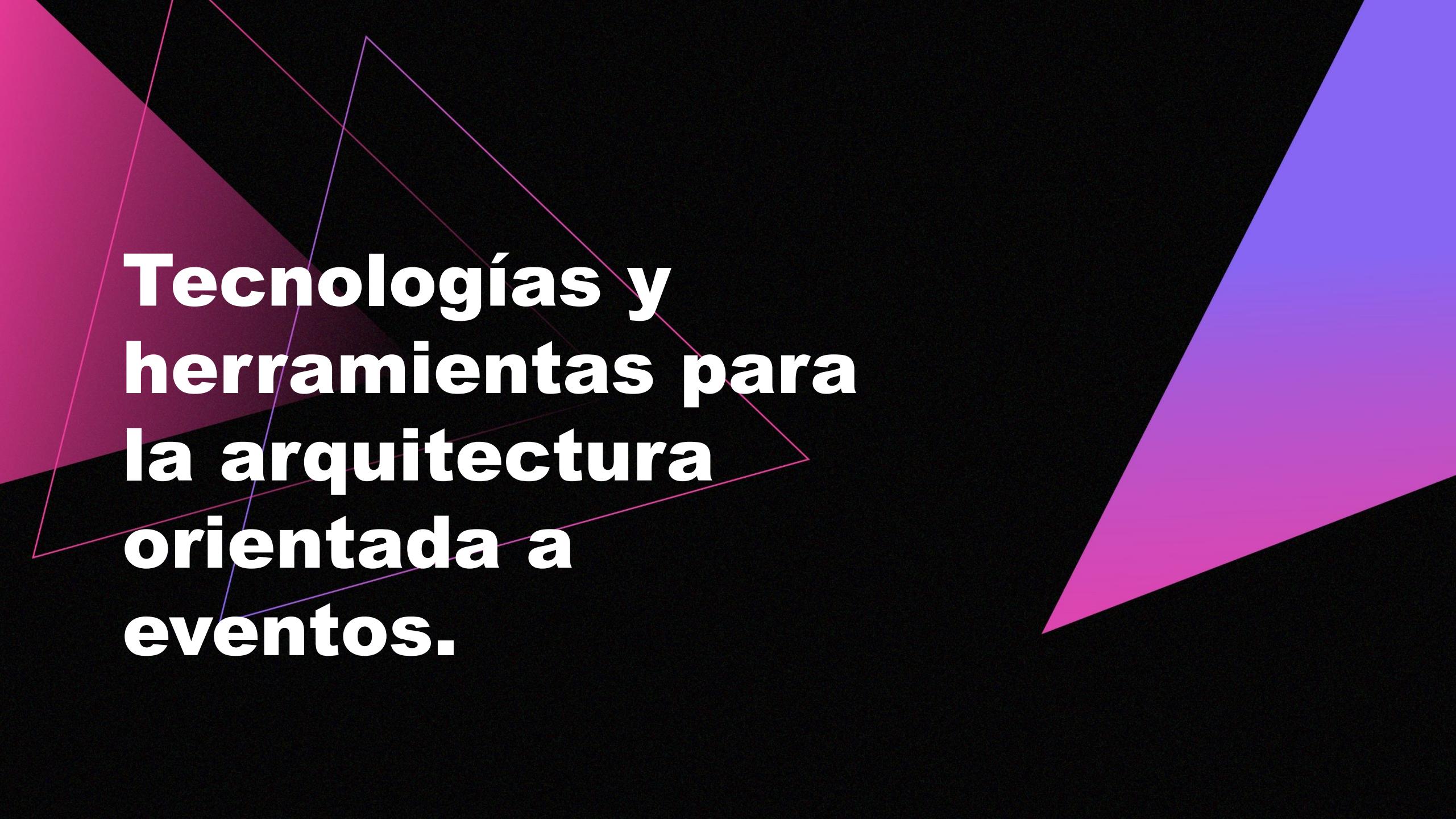
Idempotencia:

- Un **operación idempotente** produce el mismo resultado sin importar cuántas veces se ejecute con los mismos datos.
- La idempotencia se utiliza para manejar reintentos y garantizar consistencia, pero no impide la ejecución redundante.

Ejemplo:

- Un servicio de pagos verifica si una transacción ya fue procesada antes de realizar cualquier operación.
- Si se recibe un mensaje duplicado, simplemente se ignora.





**Tecnologías y
herramientas para
la arquitectura
orientada a
eventos.**

Apache Kafka:

Sistema distribuido para manejar flujos de datos en tiempo real.

Características:

- Alta tolerancia a fallos.
- Almacenamiento duradero de eventos.
- Soporta grandes volúmenes de eventos.

Casos de uso:

- Procesamiento de eventos en tiempo real.
- Almacenamiento de eventos para análisis.



RabbitMQ

Mensajería basada en protocolos AMQP.

Características:

- Fácil de configurar y usar.
- Compatible con patrones Publish/Subscribe y Work Queues.



Casos de uso:

- Microservicios.
- Comunicación entre sistemas empresariales.

Amazon SNS/SQS:

Servicios de mensajería en la nube ofrecidos por AWS.

Características:

- **SNS**: Publicación y suscripción de mensajes.
- **SQS**: Colas para procesamiento asincrónico.

Casos de uso:

- Sistemas escalables en la nube.
- Integración entre servicios en AWS.



Event Store:

Base de datos optimizada para almacenar y procesar eventos.

Características:

- Soporte nativo para Event Sourcing.
- Altamente escalable.



Casos de uso:

- Auditoría.
- Reconstrucción de estados del sistema.

Apache Cassandra:

Base de datos NoSQL distribuida.

Características:

- Soporte para grandes volúmenes de datos distribuidos.
- Alta disponibilidad.

Casos de uso:

- Auditoría.
- Sistemas de streaming con alta concurrencia.

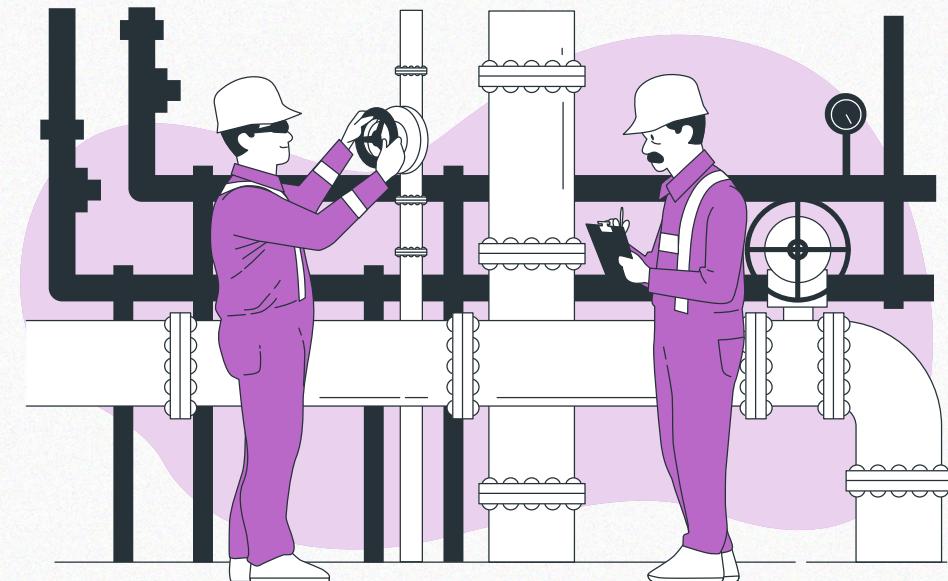


Introducción al patrón Pipe and Filter

Pipe and Filter

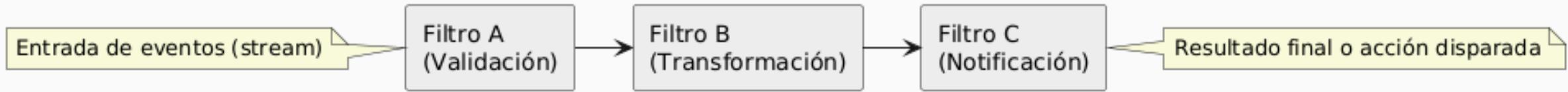
Es un patrón arquitectónico clásico donde los datos fluyen a través de una cadena de pasos independientes (filtros), conectados por canales de comunicación (pipes).

Es un patrón arquitectónico clásico donde los datos fluyen a través de una cadena de pasos independientes (filtros), conectados por canales de comunicación (pipes).



```
cat archivo.csv | grep "2024" | awk -F "," '{print $2}' | sort
```

Pipe and Filter



Filtros de foto en consola

Implementar un pipeline de filtros que transforme una imagen representada como un array de píxeles, aplicando filtros encadenados como brillo, contraste, escala de grises, etc.

Imagen → Filtro Brillo → Filtro Contraste → Filtro Sepia → Mostrar/Guardar

Si cada filtro es una transformación pura y composable, ¿no es esto exactamente un pipeline funcional?



Frameworks para desarrollo de aplicaciones basadas en eventos

Spring Cloud Stream

Framework para construir aplicaciones que procesan eventos.

Características:

- Integración con brokers como Kafka y RabbitMQ.
- Simplificación de patrones Pub/Sub.



Casos de uso:

- Aplicaciones empresariales en Java.

Spring Cloud Stream

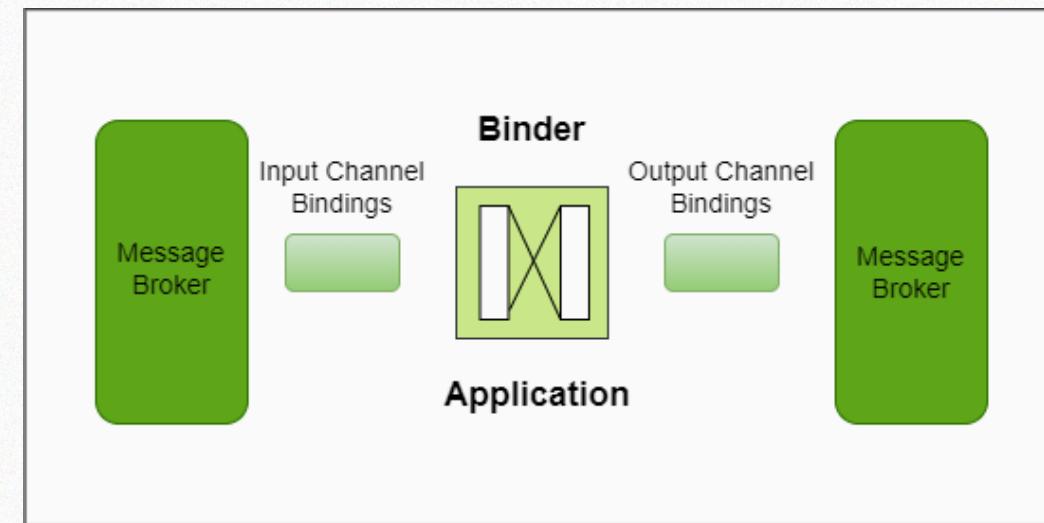
Framework para construir aplicaciones que procesan eventos.

Características:

- Integración con brokers como Kafka y RabbitMQ.
- Simplificación de patrones Pub/Sub.

Casos de uso:

- Aplicaciones empresariales en Java.



Sistema en tiempo real **(RTS)**

Un **sistema en tiempo real (RTS)** es aquel que debe cumplir con restricciones temporales estrictas, donde el “correcto funcionamiento” depende no solo de la corrección lógica de las operaciones, sino también de que se ejecuten en el tiempo requerido.

Los sistemas en tiempo real se dividen en dos categorías principales:

1. **Sistemas duros en tiempo real**: Un incumplimiento de los plazos puede causar fallos catastróficos (e.g., sistemas de control aéreo).
2. **Sistemas blandos en tiempo real**: Los retrasos son tolerables en cierta medida, pero impactan negativamente el rendimiento (e.g., reproducción de video).



Características Principales de un Sistema en Tiempo

Realismo:

La capacidad del sistema para producir un resultado predecible bajo cualquier condición.

- Ejemplo: En un sistema de control de frenos ABS, el tiempo de respuesta no debe exceder los 10 ms.

2. Confiabilidad:

Alta tolerancia a fallos debido a la criticidad de sus aplicaciones.

- Ejemplo: Sistemas médicos como marcapasos.



Aplicaciones Prácticas

1. Industria Automotriz:

- Control de frenado (ABS) y gestión del motor.
- Tiempo de respuesta crítico: 10 ms.

2. Control Aéreo:

- Gestión del tráfico aéreo en tiempo real para evitar colisiones.
- Tiempo de respuesta: milisegundos a segundos.

3. Sistemas Médicos:

- Monitoreo y control de dispositivos como ventiladores o marcapasos.



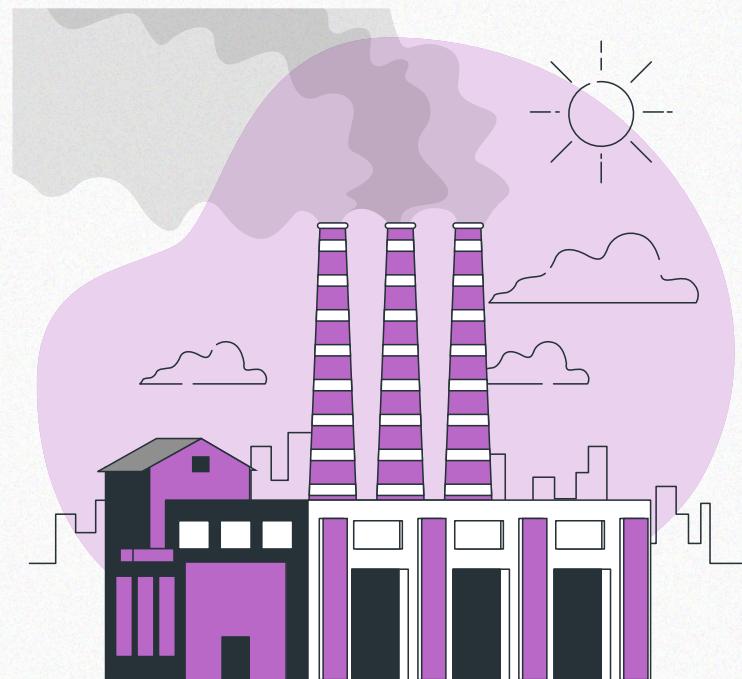
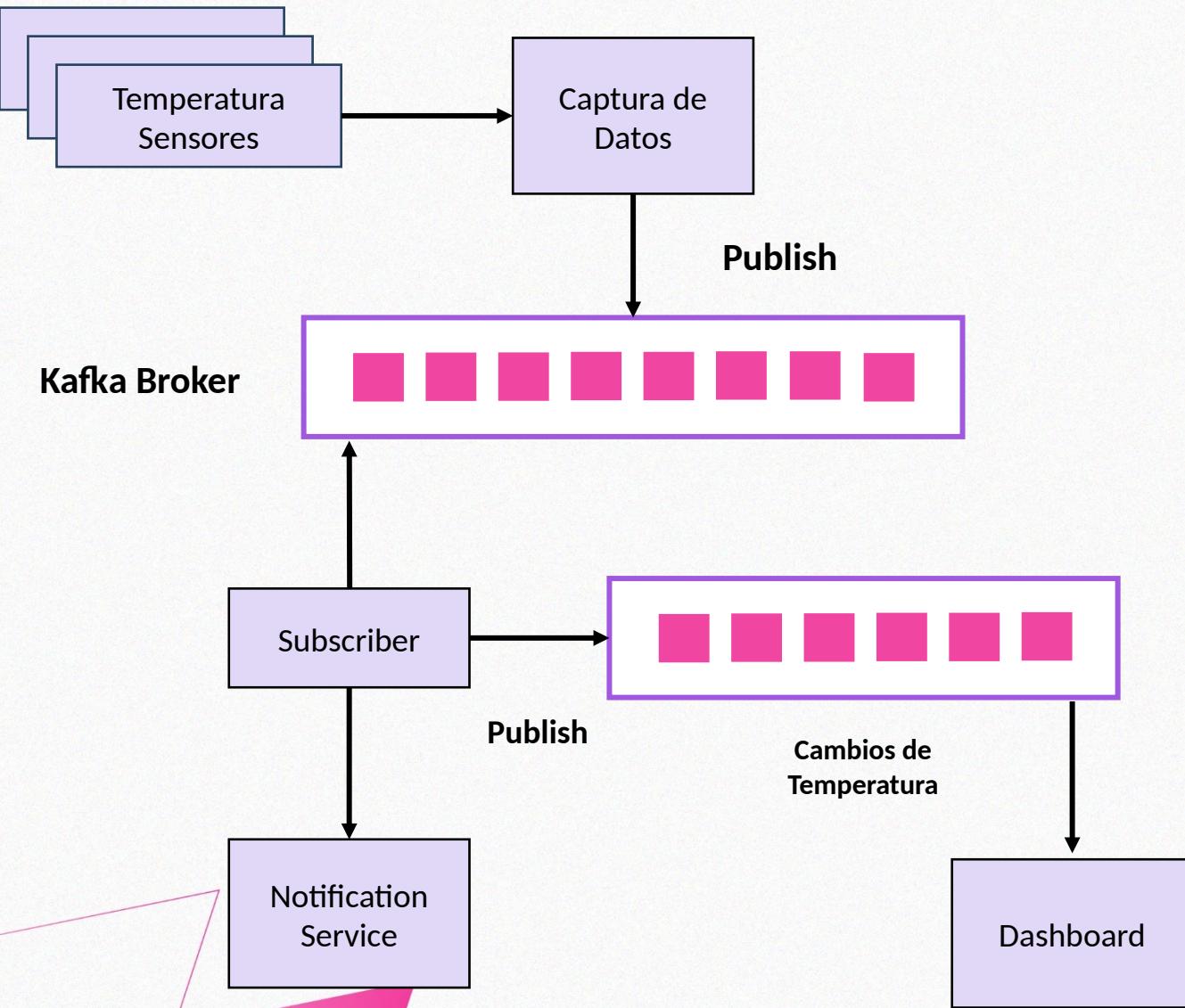
Ejemplo

Diseñar un sistema básico de monitoreo de temperatura en tiempo real para una fábrica.

Requisitos:

- Lectura de temperatura cada 100 ms.
- Generar una alerta si la temperatura supera los 70 °C.
- Registrar datos en un sistema de almacenamiento.





Actividad Interactiva

- Identificar sistemas en tiempo real en su entorno profesional.
- Clasificarlos como blandos o duros en tiempo real.

Ejemplo:

- Un sistema de videoconferencia → blando.
- Un sistema de control de un robot quirúrgico → duro.



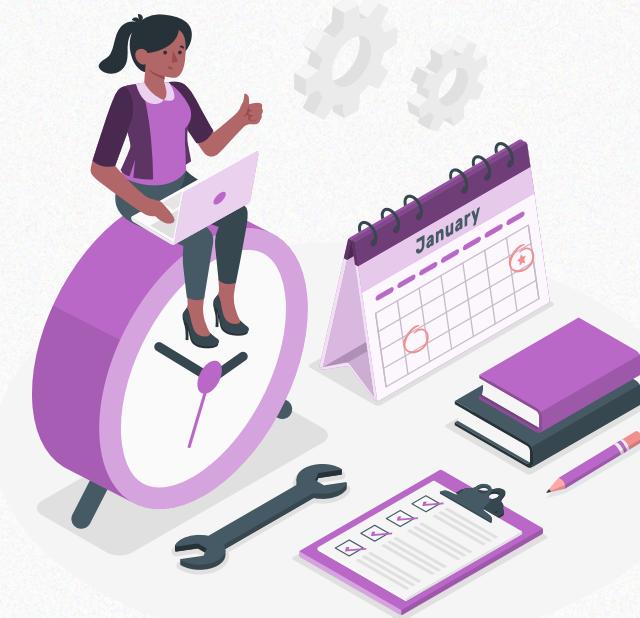
Principios Básicos del Diseño en Tiempo Real

1. Priorización de Tareas:

- Clasificar las tareas según su criticidad.
- Ejemplo: En un sistema de navegación aérea, las tareas de evitar colisiones tienen mayor prioridad que las tareas de actualización de mapas.

Métodos:

- Planificación basada en prioridades (Rate Monotonic Scheduling - RMS), Tareas con menor periodo tienen mayor prioridad.
- Planificación dinámica (Earliest Deadline First - EDF), La tarea con la fecha límite más cercana tiene prioridad.



Ejemplo

1. **T1:** Una tarea crítica que lee un sensor cada 4 ms.
2. **T2:** Un procesamiento moderado que debe realizarse cada 6 ms.
3. **T3:** Una tarea menos crítica que procesa datos cada 12 ms.

Tarea	Tiempo de ejecución (Ci)	Periodo (Ti)
T1	1 ms	4 ms
T2	2 ms	6 ms
T3	3 ms	12 ms

1.2 Modularidad y particionamiento

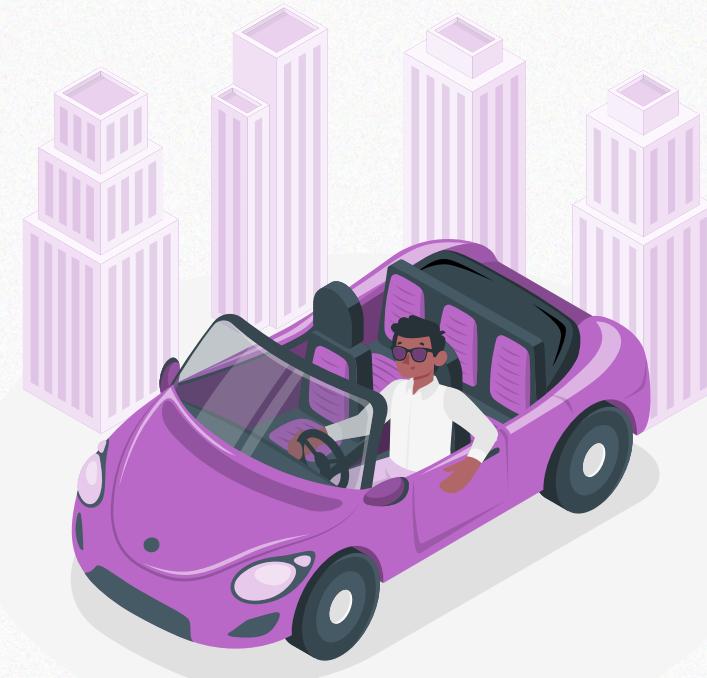
Dividir el sistema en componentes independientes.

Beneficios:

- Facilidad para realizar pruebas y mantenimiento.
- Reducción de errores al aislar responsabilidades.

Ejemplo práctico: Un sistema de control de tráfico:

- **Módulo 1:** Detección de vehículos.
- **Módulo 2:** Control de luces.
- **Módulo 3:** Generación de alertas.



1.3 Gestión de recursos

La gestión eficiente de los recursos es un aspecto crucial en el diseño de sistemas en tiempo real, ya que estos sistemas suelen operar en entornos donde los recursos (como CPU, memoria, sensores o dispositivos de E/S) son compartidos por múltiples tareas.

Solución:

- Uso de semáforos para evitar condiciones de carrera.
- Implementación de algoritmos de exclusión

Ejemplo:

- En un sistema de monitoreo, varios módulos leen un sensor de temperatura; el acceso debe estar sincronizado.



1.3 Gestión de recursos (Problemas)

1. **Condiciones de carrera:** Ocurren cuando dos o más tareas acceden simultáneamente a un recurso compartido y al menos una de ellas realiza modificaciones.

- **Ejemplo:** Dos tareas intentan actualizar un sensor al mismo tiempo, lo que genera resultados inconsistentes.

2. **Bloqueos (deadlocks):** Ocurren cuando dos o más tareas esperan indefinidamente que un recurso quede disponible.

- **Ejemplo:** Tarea A bloquea un recurso X y necesita el recurso Y, mientras que la tarea B bloquea el recurso Y y necesita el recurso X.

3. **Inversión de prioridad:** Una tarea de alta prioridad queda bloqueada porque una tarea de baja prioridad está utilizando un recurso compartido.

- **Ejemplo:** Un sistema de frenado (tarea de alta prioridad) se retrasa porque una tarea de diagnóstico (baja prioridad) está accediendo a un registro compartido.



1.3 Gestión de recursos (Soluciones)

1. **Semáforos:** Un semáforo es una variable que restringe el acceso a un recurso compartido.

- **Funcionamiento:**

- Antes de usar un recurso, una tarea realiza una operación `wait()` para verificar si está disponible.
- Una vez que termina de usar el recurso, realiza una operación `signal()` para liberarlo.

- **Ejemplo práctico:**

Un sensor de temperatura es utilizado por dos tareas. El semáforo garantiza que solo una tarea acceda al sensor en un momento dado.



1.3 Gestión de recursos

2. **Monitores:** Son mecanismos de sincronización que encapsulan recursos compartidos y las operaciones permitidas sobre ellos.

- **Ventaja:** Simplifican la programación al garantizar exclusión mutua automáticamente.
- **Ejemplo práctico:** Un monitor controla el acceso a una base de datos de registros médicos compartida por múltiples tareas.

3. **Protocolo de herencia de prioridad:** Si una tarea de alta prioridad necesita un recurso bloqueado por una tarea de baja prioridad, la tarea de baja prioridad hereda temporalmente la prioridad más alta.

- **Ejemplo práctico:** En un sistema de control de tráfico, una tarea de alta prioridad que gestiona emergencias tiene preferencia sobre las tareas de control de luces.



Patrones

Patrones Arquitectónicos

Los patrones arquitectónicos son estructuras reutilizables que proporcionan soluciones probadas para problemas comunes en el diseño de sistemas. En el caso de los sistemas en tiempo real, los patrones deben garantizar el cumplimiento de restricciones temporales, la modularidad y la escalabilidad del sistema.

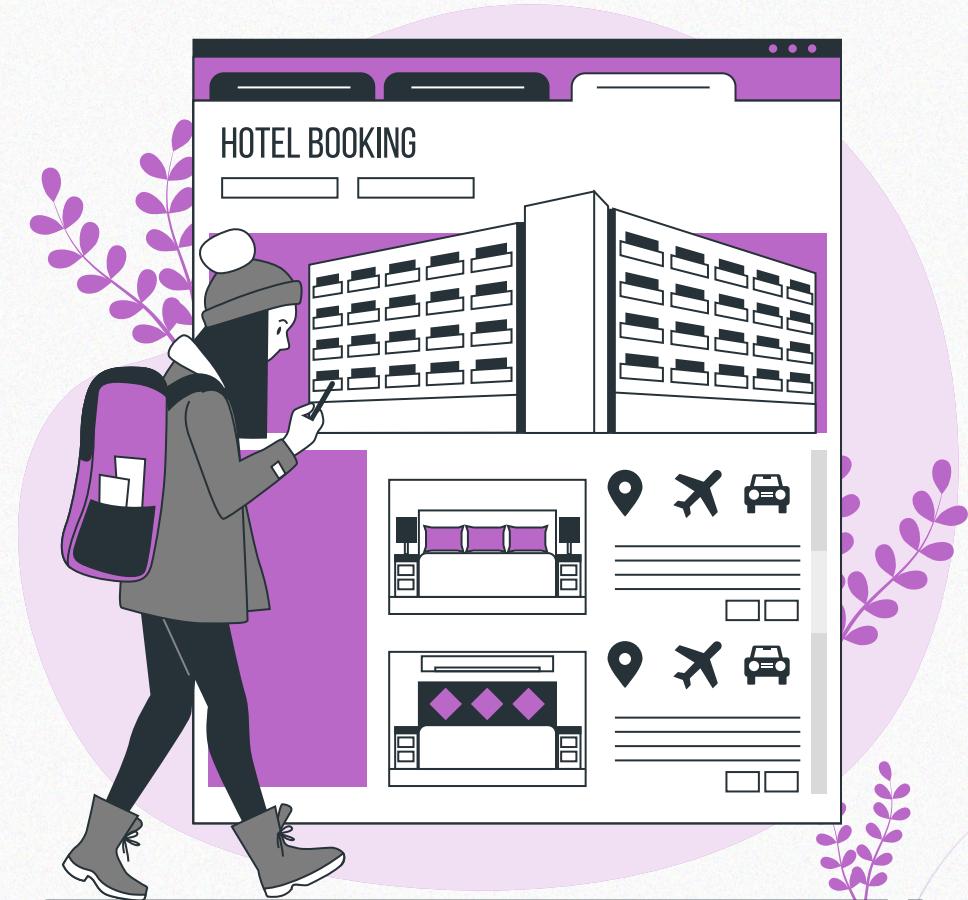


Productor-Consumidor

Este patrón separa la generación de datos (*productor*) del procesamiento de los mismos (*consumidor*) utilizando un buffer intermedio. Esto desacopla las dos etapas, permitiendo que trabajen de manera independiente.

Aplicación en tiempo real:

- En sistemas donde los datos se producen y consumen a diferentes velocidades.
- Ejemplo: Un sensor de temperatura (*productor*) genera datos que un sistema de análisis (*consumidor*) procesa.



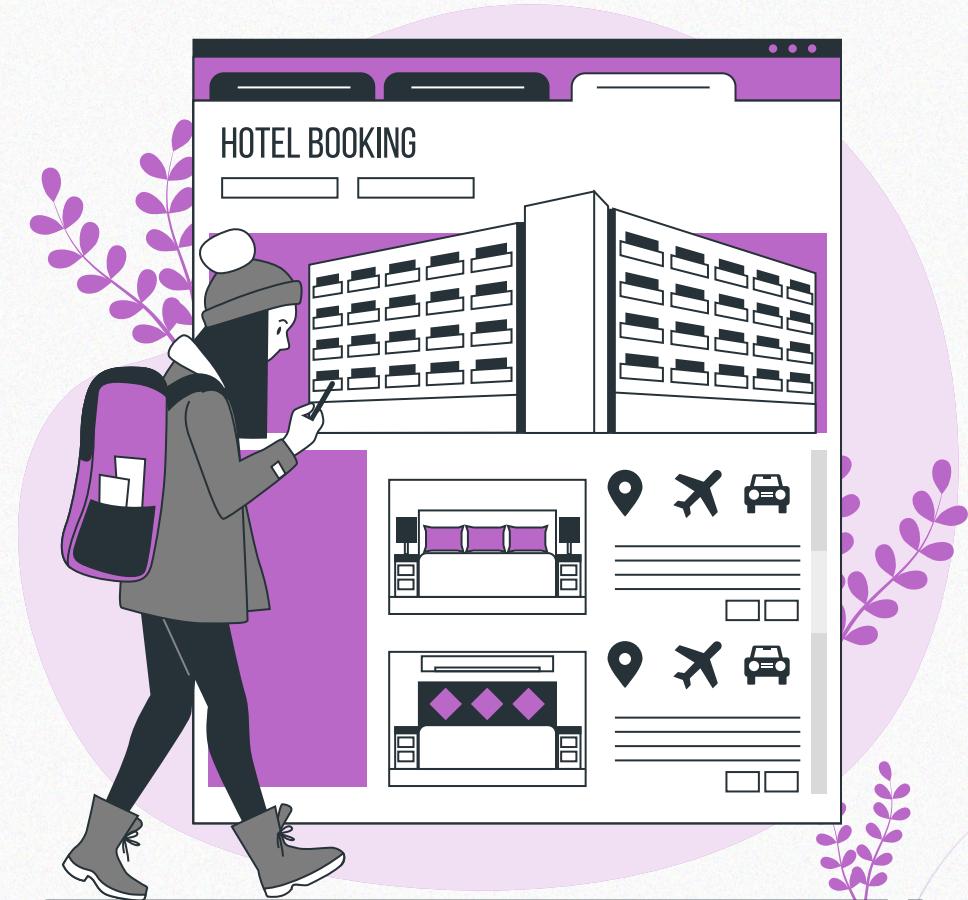
Productor-Consumidor

Ventajas:

1. Permite escalabilidad: Es posible agregar más productores o consumidores.
2. Aumenta la modularidad: El productor y el consumidor no dependen directamente uno del otro.
3. Mitiga cuellos de botella: El buffer actúa como un intermediario que regula las diferencias en la velocidad.

Consideraciones:

- **Buffer lleno:** Si el consumidor no procesa datos a tiempo, el buffer puede llenarse, causando bloqueos.
- **Buffer vacío:** Si el productor no genera datos rápidamente, el consumidor puede quedar inactivo.



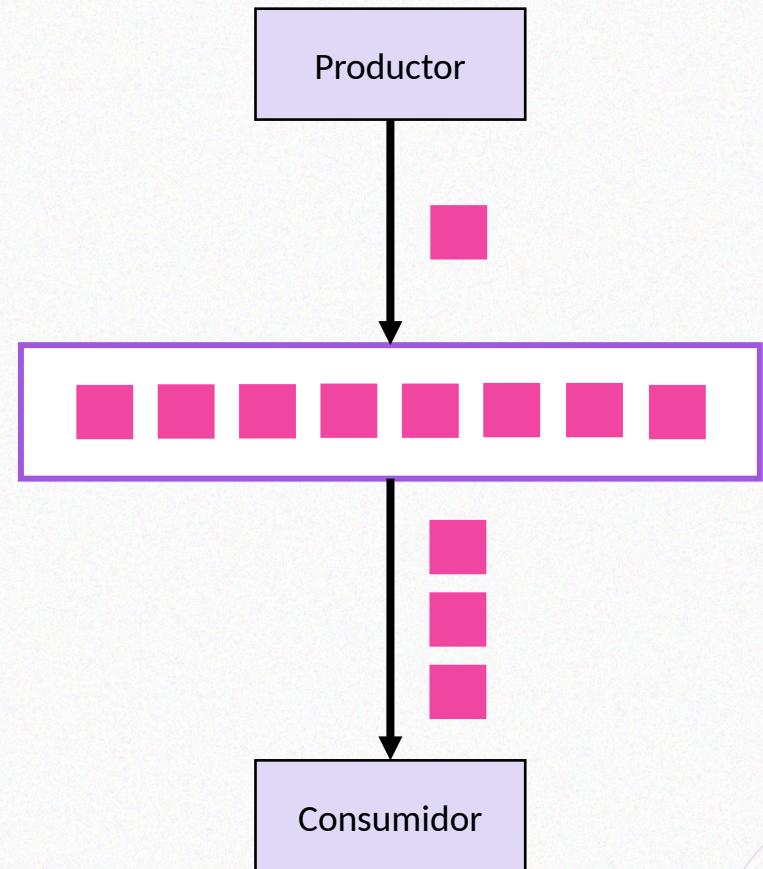
Ejemplo

Sistema de Monitoreo de Temperatura

1. **Productor:** Un sensor genera datos de temperatura cada 100 ms.
2. **Buffer:** Un buffer FIFO (First-In, First-Out) almacena hasta 10 valores.
3. **Consumidor:** Un módulo de análisis consume datos del buffer cada 200 ms para calcular promedios.

Flujo del patrón:

1. El productor genera un valor de temperatura y lo coloca en el buffer.
2. El consumidor verifica si hay datos disponibles en el buffer.
3. El consumidor procesa el dato más antiguo y lo elimina del buffer.



Reactor

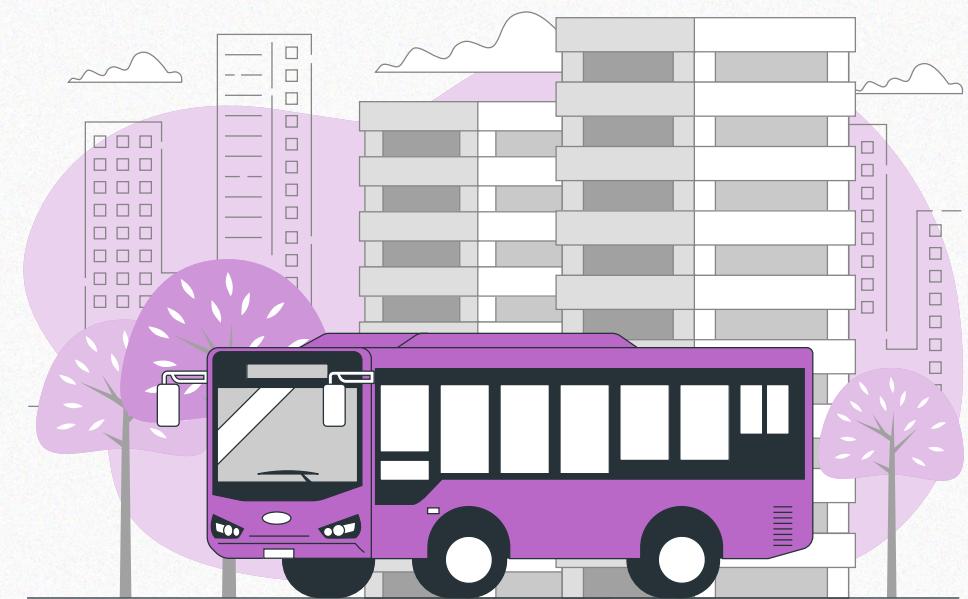
El patrón Reactor gestiona múltiples eventos concurrentes en un único bucle de eventos. Es eficiente para manejar sistemas con gran cantidad de entradas/salidas (I/O).

Aplicación en tiempo real:

- Ideal para servidores o sistemas que manejan múltiples conexiones simultáneamente.
- Ejemplo: Un servidor de red que procesa solicitudes de varios clientes en tiempo real.

Ventajas:

1. Ahorra recursos: Requiere un único hilo o proceso para manejar múltiples eventos.
2. Reduce la complejidad: Los eventos se procesan en un bucle centralizado.



Reactor

1. Bucle de Eventos (Event Loop):

Es el núcleo del patrón.

- Monitorea eventos en las fuentes registradas (e.g., lectura de datos, interrupciones de hardware).
- Delega el procesamiento de cada evento a un controlador correspondiente.

2. Descriptores de Eventos:

Representan las fuentes de eventos.

- **Ejemplo:** Un socket de red, un archivo en espera de ser leído, o una señal del sistema operativo.

3. Demultiplexor:

Gestiona la interacción entre el bucle de eventos y los descriptores.

- **Ejemplo:** select(), poll(), o epoll() en sistemas basados en Unix.

4. Controladores de Eventos (Event Handlers):

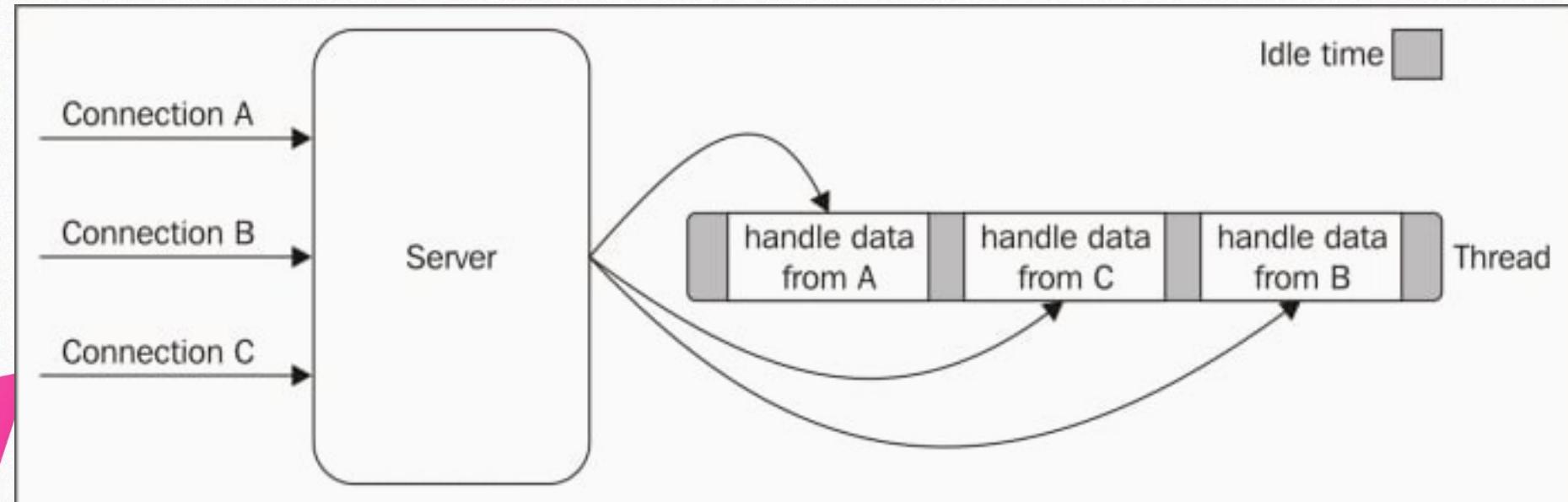
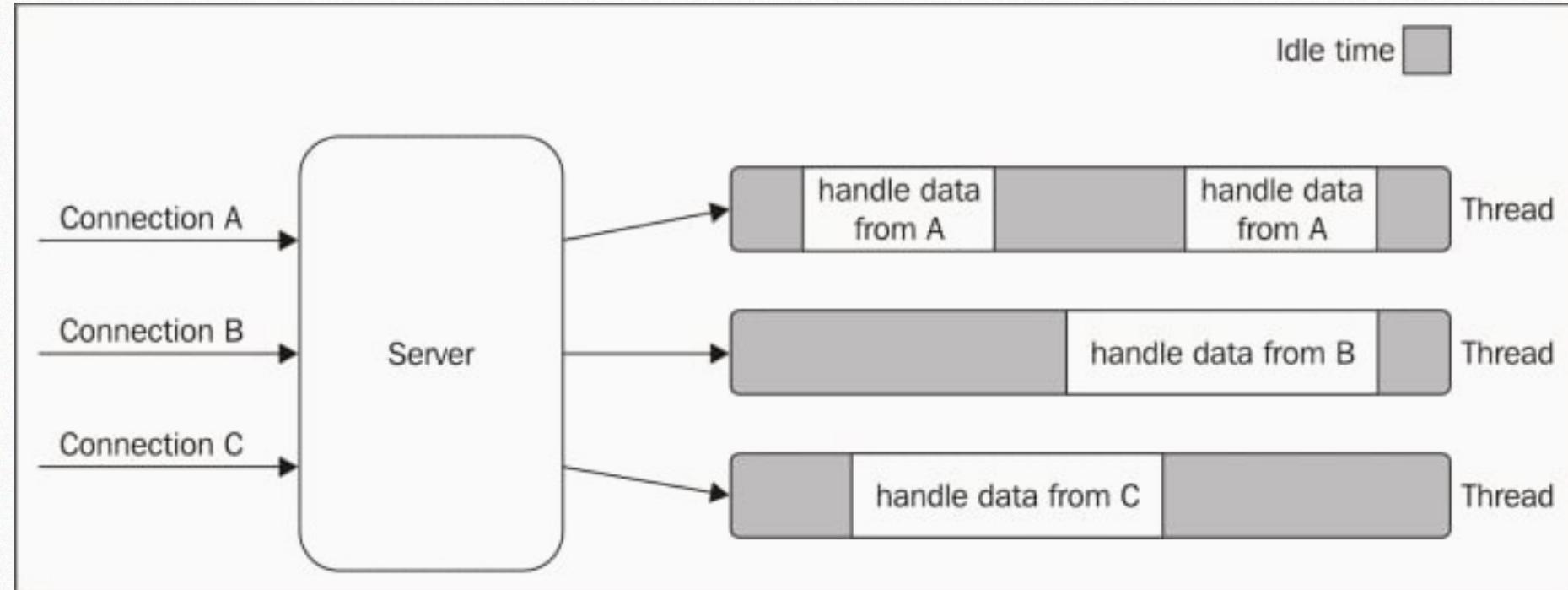
Contienen la lógica para procesar los eventos específicos.

- Cada controlador se asocia con un descriptor de evento.

5. Reactor:

Es la implementación del patrón, combinando el bucle de eventos y los controladores.

Reactor



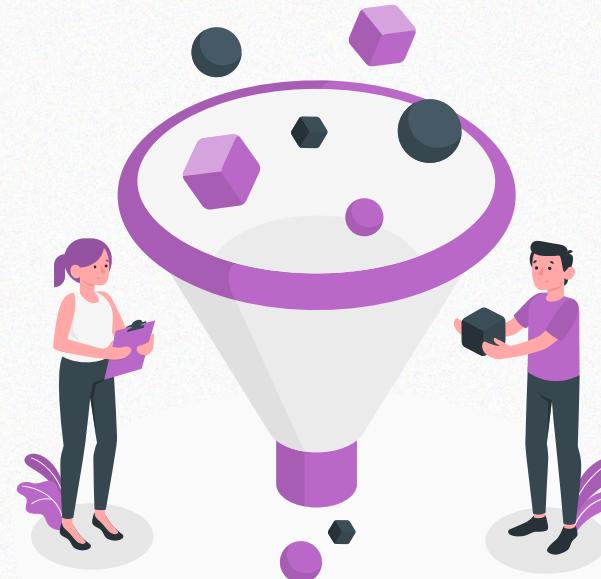
Pipe/Filter

El patrón Pipe/Filter organiza el procesamiento de datos como una serie de pasos independientes (filtros), conectados por canales (pipes).

- Ideal para sistemas con flujos de datos complejos que requieren múltiples transformaciones o análisis.
- Ejemplo: Un sistema de video en tiempo real que aplica filtros para detección de movimiento, compresión y transmisión.

Ventajas:

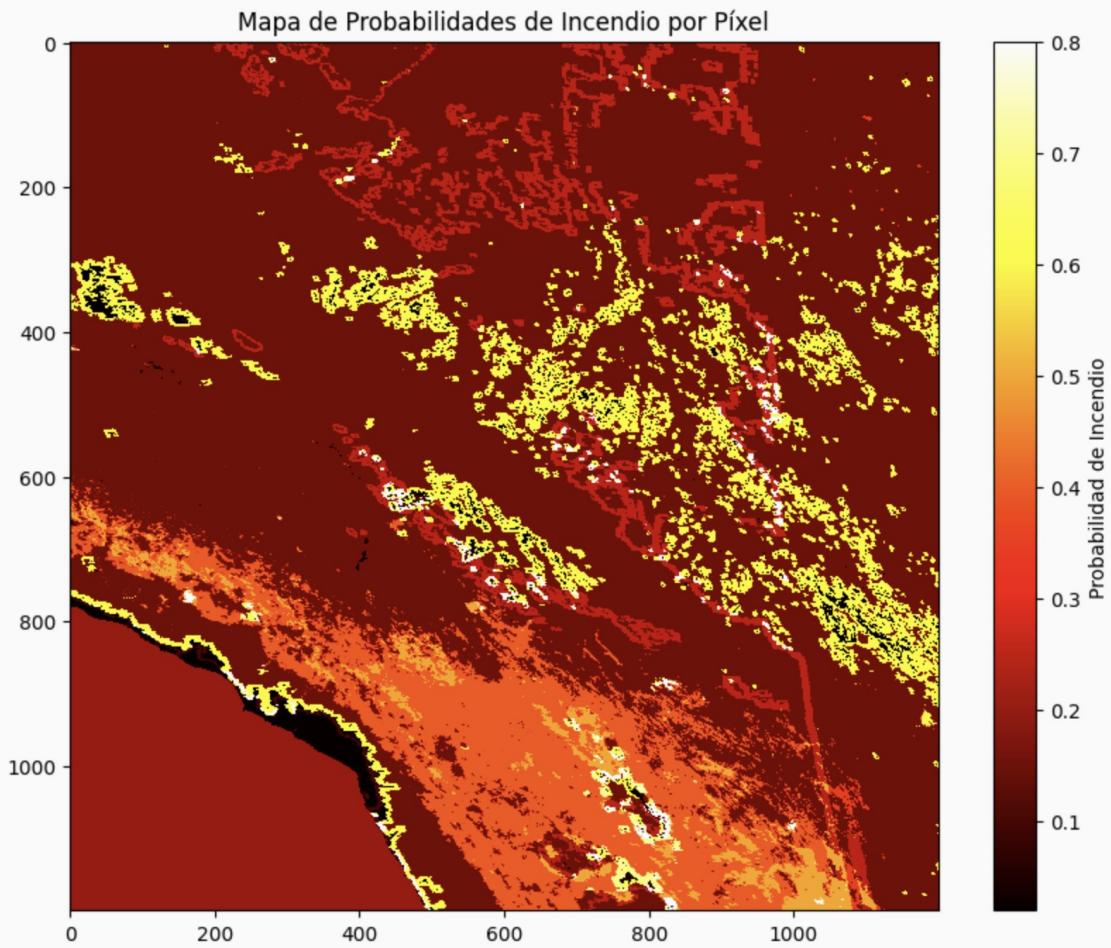
1. Modularidad: Cada filtro tiene una única responsabilidad.
2. Flexibilidad: Es fácil agregar, reemplazar o reorganizar filtros.
3. Paralelismo: Los filtros pueden ejecutarse simultáneamente para optimizar el rendimiento.



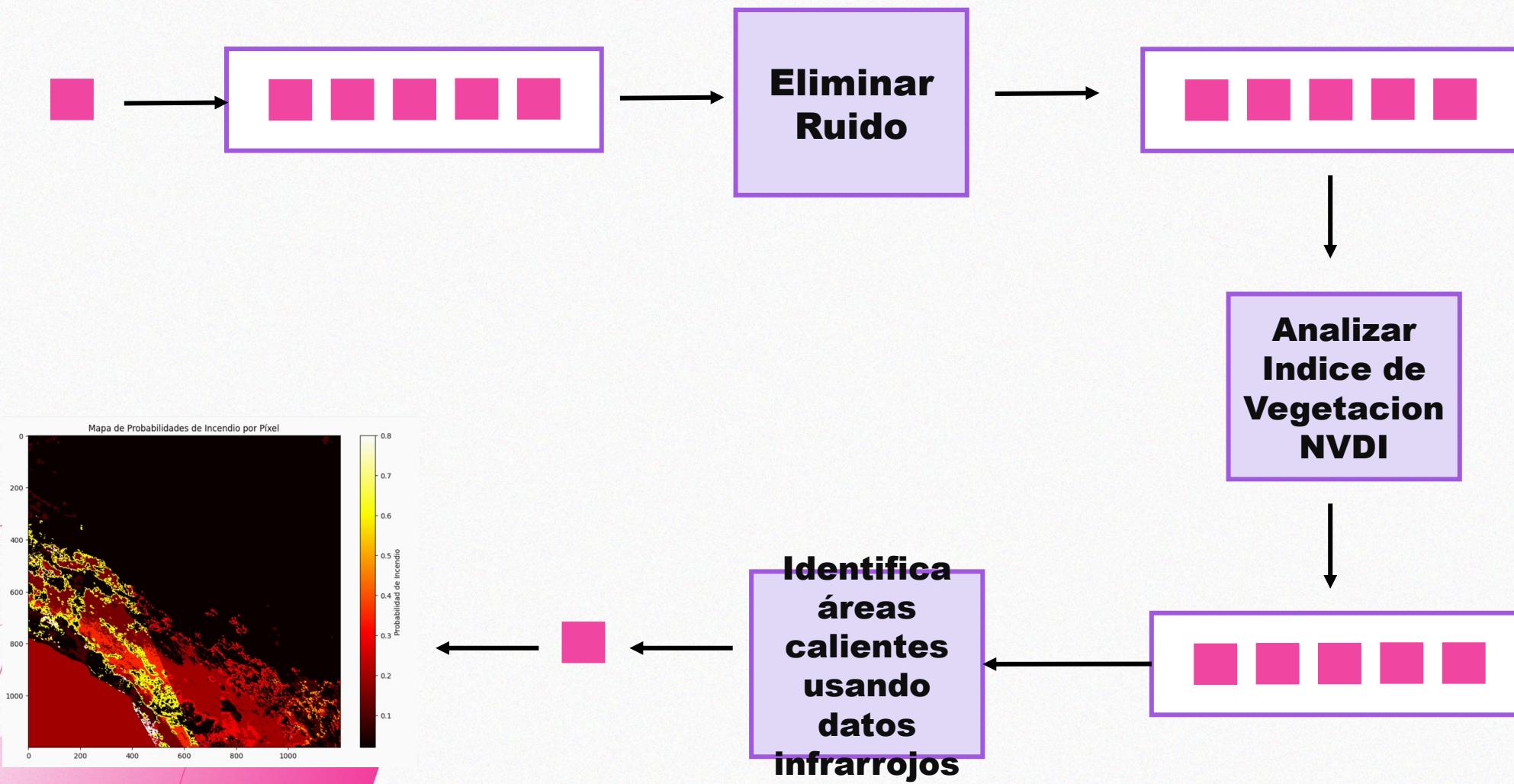
Ejemplo

Sistema de Detección de Incendios Forestales

1. **Fuente:** Imágenes satelitales (datos de entrada).
2. **Filtros:**
 - **Filtro 1:** Elimina ruido de las imágenes.
 - **Filtro 2:** Analiza el índice de vegetación (NDVI).
 - **Filtro 3:** Identifica áreas calientes usando datos infrarrojos.
3. **Pipe:** Los datos fluyen entre los filtros y finalmente generan un mapa de riesgo.



Ejemplo



Ejemplo

```
# Generador de datos (simula un sensor de temperatura)
def sensor_data():
    for _ in range(20): # Genera 20 lecturas de temperatura
        yield random.uniform(-10, 50) # Temperaturas entre -10°C y 50°C

# Filtro 1: Filtra datos fuera del rango válido
def filter_invalid(data_stream):
    for temperature in data_stream:
        if -5 <= temperature <= 45: # Rango válido
            yield temperature
        else:
            print(f"[Filtro 1] Dato descartado: {temperature:.2f}°C (fuera de rango)")

# Filtro 2: Normaliza las temperaturas (escalando entre 0 y 1)
def normalize_data(data_stream):
    min_temp, max_temp = -5, 45
    for temperature in data_stream:
        normalized = (temperature - min_temp) / (max_temp - min_temp)
        yield normalized

# Filtro 3: Genera alertas si la temperatura es alta
def alert_on_high_temp(data_stream):
    threshold = 0.8 # 80% del rango normalizado
    for normalized_temp in data_stream:
        if normalized_temp > threshold:
            print(f"[Filtro 3] Alerta: Temperatura crítica detectada ({normalized_temp:.2f})")
        else:
            yield normalized_temp
```

Ejemplo

```
# Salida final: Imprime los resultados procesados
def output_results(data_stream):
    for normalized_temp in data_stream:
        print(f"[Salida] Temperatura procesada: {normalized_temp:.2f} (normalizada)")

# Flujo del sistema usando Pipe/Filter
if __name__ == "__main__":
    print("Inicio del procesamiento de datos de temperatura...\n")

    data_stream = sensor_data()                                # Fuente de datos
    data_stream = filter_invalid(data_stream)                  # Aplicar Filtro 1
    data_stream = normalize_data(data_stream)                 # Aplicar Filtro 2
    data_stream = alert_on_high_temp(data_stream)             # Aplicar Filtro 3
    output_results(data_stream)                               # Salida final

    print("\nProcesamiento completado.")
```

Evaluación de Arquitecturas en Tiempo Real

La evaluación de arquitecturas en tiempo real es un paso crucial para garantizar que un sistema cumpla con las restricciones temporales, utilice los recursos eficientemente y sea confiable bajo todas las condiciones operativas.

- 1. Garantizar el cumplimiento de plazos:** Las tareas críticas deben completarse dentro de los tiempos límite definidos.
- 2. Maximizar la utilización de recursos:** Evitar el uso excesivo o ineficiente de CPU, memoria y red.
- 3. Probar la confiabilidad:** Asegurarse de que el sistema siga funcionando correctamente incluso ante fallos.
- 4. Validar la escalabilidad:** Verificar que el sistema pueda manejar un aumento en la carga sin degradar el rendimiento.



Criterios de Evaluación

Tiempo de Respuesta: El tiempo de respuesta es el lapso desde que un evento ocurre (e.g., datos de un sensor) hasta que el sistema lo procesa completamente.

Ejemplo:

En un sistema de frenado ABS:

- Detectar la velocidad de rotación de las ruedas: **1 ms**.
- Calcular la presión de frenado: **5 ms**.
- Aplicar la presión: **2 ms**.
- Tiempo total: **8 ms** (cumple con el límite de 10 ms).



Métodos de Evaluación:

- **Análisis estático:** Calcular el tiempo de ejecución en el peor caso (**WCET**).
- **Pruebas prácticas:** Ejecutar tareas críticas y medir su tiempo en condiciones reales.
- **Simulación:** Realizar pruebas de carga que permitan simular un escenario de alta densidad de solicitudes.



Edge Computing

El edge computing permite el procesamiento de datos en la periferia de la red, más cerca de las fuentes de datos, para reducir los tiempos de respuesta y el tráfico hacia servidores centrales



Ventajas

1. Latencia reducida:

- Procesar datos localmente evita retrasos asociados a la transferencia hacia servidores centrales.
- Ejemplo: Un sistema de control de tráfico procesando datos de sensores en el borde puede cambiar luces en milisegundos.

2. Confiabilidad:

- Si la conectividad a la nube falla, el procesamiento local asegura que el sistema siga funcionando.
- Ejemplo: Un dron que realiza cálculos a bordo no depende de una conexión remota para tomar decisiones críticas.

3. Ahorro de ancho de banda:

- Solo los datos relevantes se envían a la nube, lo que reduce el tráfico de red.

4. Privacidad y seguridad:

- Procesar datos localmente minimiza el riesgo de exposición de datos sensibles.

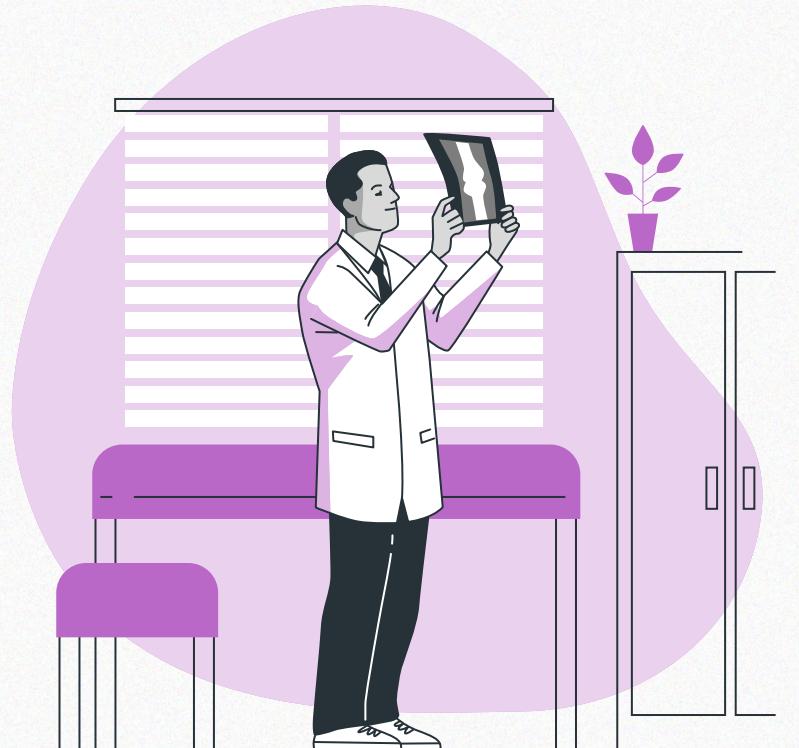
Casos de Uso Críticos en Tiempo Real

. Vehículos Autónomos:

- Los cálculos relacionados con detección de obstáculos, navegación y frenado deben realizarse en tiempo real a bordo del vehículo.
- **Latencia aceptable:** Milisegundos.

2. Sistemas Médicos:

- Monitores portátiles de pacientes procesan datos vitales en dispositivos edge, generando alertas inmediatas.
- **Latencia aceptable:** Menos de 500 ms.



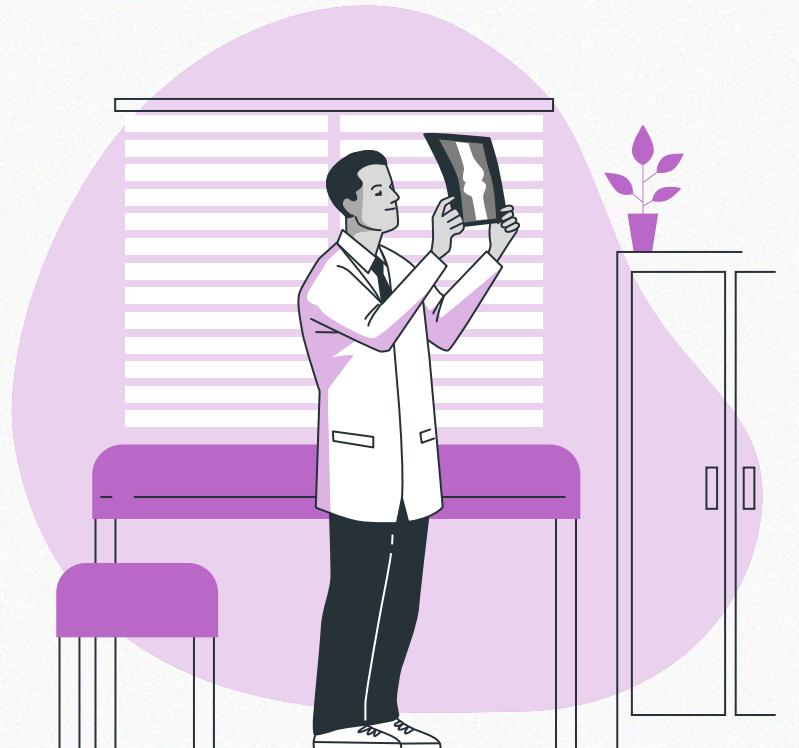
Casos de Uso Críticos en Tiempo Real

. Vehículos Autónomos:

- Los cálculos relacionados con detección de obstáculos, navegación y frenado deben realizarse en tiempo real a bordo del vehículo.
- **Latencia aceptable:** Milisegundos.

2. Sistemas Médicos:

- Monitores portátiles de pacientes procesan datos vitales en dispositivos edge, generando alertas inmediatas.
- **Latencia aceptable:** Menos de 500 ms.



NVIDIA Jetson

```
# Configurar el modelo de detección
net = detectNet("ssd-mobilenet-v2", threshold=0.5)

# Fuente de video (cámara)
camera = videoSource("/dev/video0")
output = videoOutput("display://0") # Mostrar resultados

# Bucle de procesamiento
while True:
    frame = camera.Capture() # Capturar imagen
    detections = net.Detect(frame) # Detectar objetos
    output.Render(frame) # Mostrar imagen procesada
```